Delft Students on Software Architecture
DESOSA 2016

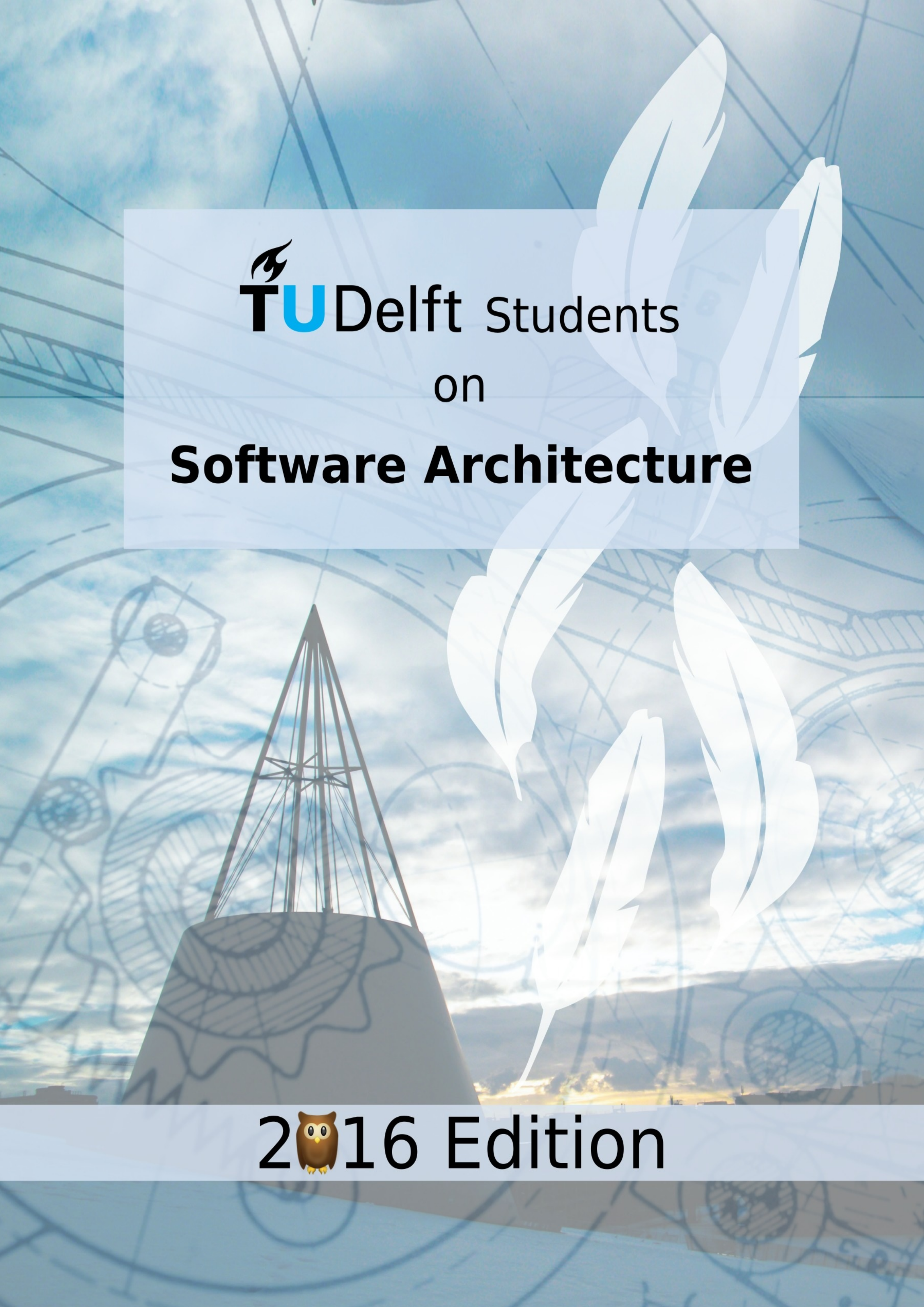
van Deursen, Arie; Aniche, M.; Aué, Joop

DOI
10.5281/zenodo.57924

Publication date
2016

Document Version
Final published version

Citation (APA)
van Deursen, A., Aniche, M., & Aué, J. (2016). *Delft Students on Software Architecture: DESOSA 2016*. Delft University of Technology. https://doi.org/10.5281/zenodo.57924



TU Delft Students on Software Architecture

2🦉16 Edition

Table of Contents

Delft Students on Software Architecture: DESOSA 2016

Arie van Deursen, **Maurício Aniche** and **Joop Aué**.

Delft University of Technology, The Netherlands, May 11, 2016, Version 1.1

We are proud to present *Delft Students on Software Architecture*, a collection of 21 architectural descriptions of open source software systems written by students from Delft University of Technology during a master-level course taking place in the spring of 2016.

In this course, teams of 3-4 students could adopt a project of choice on GitHub. The projects selected had to be sufficiently complex and actively maintained (one or more pull requests merged per day). The systems are from a wide range of domains, including web development (Ember, D3, Ruby on Rails), machine learning (TensorFlow), music (SonicPi), and gaming (Terasology).

During a 10 week period, the students spent one third of their time on this course, and engaged with these systems in order to understand and describe their software architecture.

Inspired by Brown and Wilsons' Architecture of Open Source Applications, we decided to organize each description as a chapter, resulting in the present online book.

This book is the second in volume the DESOSA series: The first DESOSA book resulted from the 2015 edition of the course, and contained architectural descriptions of ten (different) open source systems.

Recurring Themes

The chapters share several common themes, which are based on smaller assignments the students conducted as part of the course. These themes cover different architectural 'theories' as available on the web or in textbooks. The course used Rozanski and Woods' Software Systems Architecture, and therefore several of their architectural viewpoints and perspectives recur.

The first theme is outward looking, focusing on the use of the system. Thus, many of the chapters contain an explicit stakeholder analysis, as well as a description of the context in which the systems operate. These were based on available online documentation, as well as on an analysis of open and recently closed issues for these systems.

A second theme involves the development viewpoint, covering modules, layers, components, and their inter-dependencies. Furthermore, it addresses integration and testing processes used for the system under analysis, and often includes an assessment of the presence of technical debt.

A third recurring theme is *variability management*. Many of today's software systems are highly configurable. In such systems, different features can be enabled or disabled, at compile time (flags) or at run time (toggles). Using techniques from the field of product line engineering, several of the chapters provide feature-based variability models of the systems under study.

First-Hand Experience

Last but not least, the chapters are also based on the student's experience in actually contributing to the systems described. As part of the course over 75 pull requests to the projects under study were made, including refactorings (Ember 13088, Rails 24198), bug fixes (Terasology 2235), new features (Karma 1983), test cases (Sonic-Pi 1054), translations (OpenTripPlanner 2232), and documentation (OpenCV 4375). Many projects had issues explicitly marked as "good for new contributors", making it easier for students to find a good starting point.

Through these contributions the students often interacted with lead developers and architects of the systems under study, gaining first-hand experience with the architectural trade-offs made in these systems.

Feedback

While we worked hard on the chapters to the best of our abilities, there will be plenty of omissions and inaccuracies. We value your feedback on any of the material in the book. For your feedback, you can:

- Open an issue on our GitHub repository for this book.
- Offer an improvement to a chapter by posting a pull request on our GitHub repository.
- Contact @delftswa on Twitter.
- Send an email to Arie.vanDeursen at tudelft.nl.

Acknowledgments

We would like to thank:

- Our guest speakers: Alex Nederlof, Felienne Hermans, Johan den Haan, Tobias Kuipers, Huub Bakker, Hans van Dongen, Daniele Romano, Maikel Lobbezzo.
- Valentine Mairet who created the front cover of this book.
- Michael de Jong and Alex Nederlof who were instrumental in the earlier editions of this course.
- All open source developers who helpfully responded to the students' questions and contributions.
- The excellent gitbook toolset and gitbook hosting service making it easy to publish a collaborative book like this.

Further Reading

1. Arie van Deursen, Alex Nederlof, and Eric Bouwers. Teaching Software Architecture: with GitHub! avandeursen.com, December 2013.
2. Nick Rozanski and Eoin Woods. Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives. Addison-Wesley, 2012, 2nd edition.
3. Sven Apel, Don Batory, Christian Kästner, Gunter Saake. Feature-Oriented Software Product Lines: Concepts and Implementation. Springer-Verlag, 2013.
4. Eric Bouwers. Metric-based Evaluation of Implemented Software Architectures. PhD Thesis, Delft University of Technology, 2013.
5. Amy Brown and Greg Wilson (editors). The Architecture of Open Source Applications. Volumes 1-2, 2012.
6. Arie van Deursen and Rogier Slag (editors). Delft Students on Software Architecture: DESOSA 2015. delftswa.github.io, 2015.

Copyright and License

Cover image credits: TU Delft library, TheSpeedX at Wikimedia; Owl on Emojipedia Sample Image Collection at Emojipedia; Feathers by Franco Averta at Flaticon.

Atom - the hackable text editor

By Rowan Bottema, Ruben Koeze, Robbert van Staveren and Freek van Tienen
Delft University of Technology

Abstract

Atom is an open source hackable text editor, developed by GitHub but also by other contributors. In this chapter everything concerning Atom in the development and testing is dissected. First, all stakeholders involved in or with Atom will be discussed. After this, the structure and organization of Atom is dissected: how Atom is build and what methods are used. In the third part, an overview of all features that can extend Atom is given, as well as the relationships between these features. Lastly, in the functional view, all the elements delivering the functionalities of Atom are broken down into pieces.

Table of contents

Introduction

Atom is an open source text editor, designed to be hackable to the core. It started as a side project of Github founder @defunkt in 2008, then called Atomicio[1]. Because he was mainly busy with the GitHub platform, Atomicio was put on hold for a couple of years. Then in 2011, when GitHub was already getting bigger, they upgraded it to an official GitHub project called Atom. Currently, it has more than a million active users and over 28,000 commits have been committed to the main repository, with many more committed to related repositories. It is mainly developed by the company GitHub, but at the same time almost 300 people contributed to the main repository, indicating a big open source community. These aspects make the Atom architecture an interesting object of study.

Stakeholders

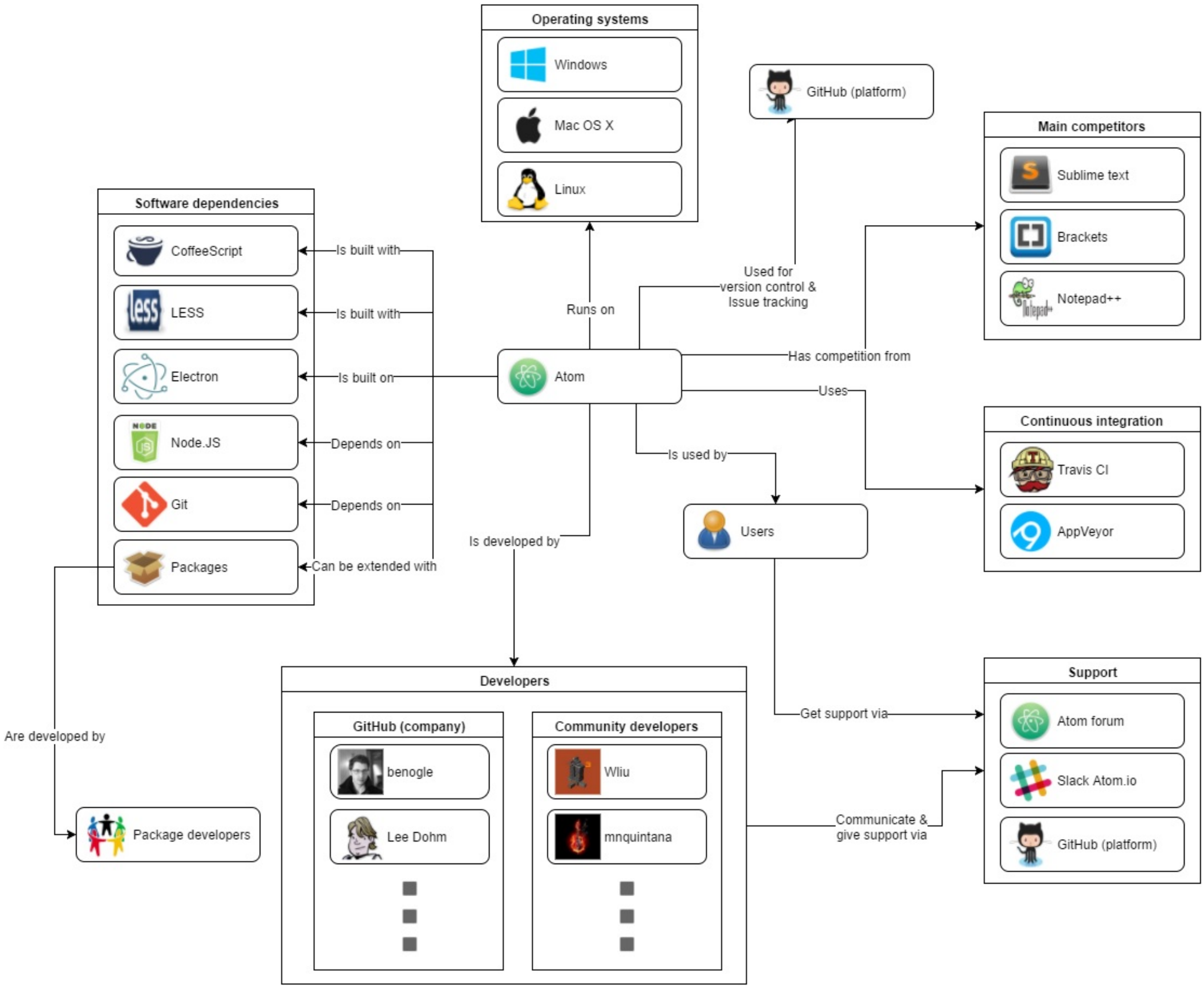
Because Atom is an open source project and evolved a lot over time, there are many stakeholders involved in the project. Below is an overview of the most important stakeholders from *Rozansky and Woods*[2]:

- **Developers**: Consist mainly of GitHub Staff, but there are other active persons not related to GitHub who also contribute a lot to the Atom project. The GitHub staff wants to invite new developers into the project and gives them a lot of support. One of the ways they try to get this done by giving some issues the “beginner” label, which means the issue is simple enough for a beginning developer to pick up. It is interesting to note that Facebook also holds stakes in Atom, as they develop their Nuclide IDE on top of Atom. This is evidenced by 4 of the 5 top contributors of Nuclide being in the top 50 contributors of Atom[6].
- **Users**: Consist mainly of highly advanced computer users like programmers and system administrators. Atom is also preferred by users when an IDE is not available or too much work to set up.
- **Maintainers**: Atom is maintained by the core developers, but GitHub is the main driver in keeping the project alive. They make sure that there are enough developers for the Atom project and invest time and money in making sure the Atom project stays alive.
- **Support staff**: Consists mainly of a lot of volunteers, but next to that also the GitHub staff and core developers. This is done by responding to issues, triaging issues and pull requests, closing bugs as duplicate or filed on the wrong repository, and much more. Also the Forum and Slack are used for quick support to the end users and developers. Lee Dohm is the main community manager and manages all supporting staff.
- **Testers**: The main users test the system and report bugs to the issue tracker on GitHub. There is no special test team available although the core developers test most pull requests before merging.

- **Production Engineers**: All tools like GitHub, the Forum, Slack and the build servers are maintained by the GitHub team. They make sure it is running and invest time in updates or setting up new tools.
- **Assessors**: The core development team of Atom oversees the conformance of the programming standards. They check each pull request with multiple people and also check if it is still in line with the future planning.

Context

To get an overview of what is happening with and around Atom, a context view is created. The context view shown below will be shortly discussed.



Atom's context view

Software dependencies

Atom depends on a few technologies, shown in the *Software dependencies* box. Atom is built in CoffeeScript and LESS, which is compiled to JavaScript and CSS [3]. All this is built on top of Electron, a Node.js package that interprets and runs these languages. Last but not least, Git can be used directly from the editor, and is therefore bundled with the editor.

Developers

Atom is being developed by multiple parties. These will be discussed in the next paragraphs.

Main Developer: GitHub

It is important to notice GitHub's role in the development of Atom. As shown in the Context View, GitHub is used for version control & issue tracking. This is the platform, or website, GitHub.com. In addition to the platform, GitHub is also mentioned as a company.

Other developers

A lot of different people that do not work at GitHub are working on Atom. Most of these people are not constant contributors, but two people could be identified as main contributors. Wliu and mnquintana are active in numerous issues and pull requests. Important to notice is that these two are not the only contributors, but that these serve merely as example.

Main competitors

There are some important differences between Atom and its main competitors competitors.

- **Sublime text:** Sublime text is not an open source editor, unlike Atom. And even though Sublime can be used free, they urge you to pay licensing fee. Sublime is similar to Atom in that it is extendable in much the same way as Atom.
- **Notepad++:** Notepad++ is a widely used text editor which exists since 2003. The main "problem" with Notepad++ is, that it is Windows only, whereas Atom can be used on Windows, Mac and Linux.
- **Brackets:** Brackets is most like Atom. It is open source, and also written in the web languages HTML, CSS and JavaScript. The main difference with Atom is that the focus of Brackets lies on web development only, whereas Atom does not focus on one programming field in particular.

Support

Support of Atom is given via multiple channels. The feedback is given by GitHub and other developers. The main platform for getting feedback is the Atom forum. Atom tried to switch to Slack as main support platform. After discovering that in Slack only 10.000 messages can be read back when using the free version, the decision was made to use the forum and Slack side by side. Finally, GitHub is used for issue tracking and getting feedback.

Continuous integration

Two tools are used for continuous integration on GitHub: Travis CI and AppVeyor Travis CI is used for testing if the build runs on Linux and Mac OS. AppVeyor is the CI client for running the build test on Windows. Important to notice is that Travis CI also runs the specs, but only on Mac OS.

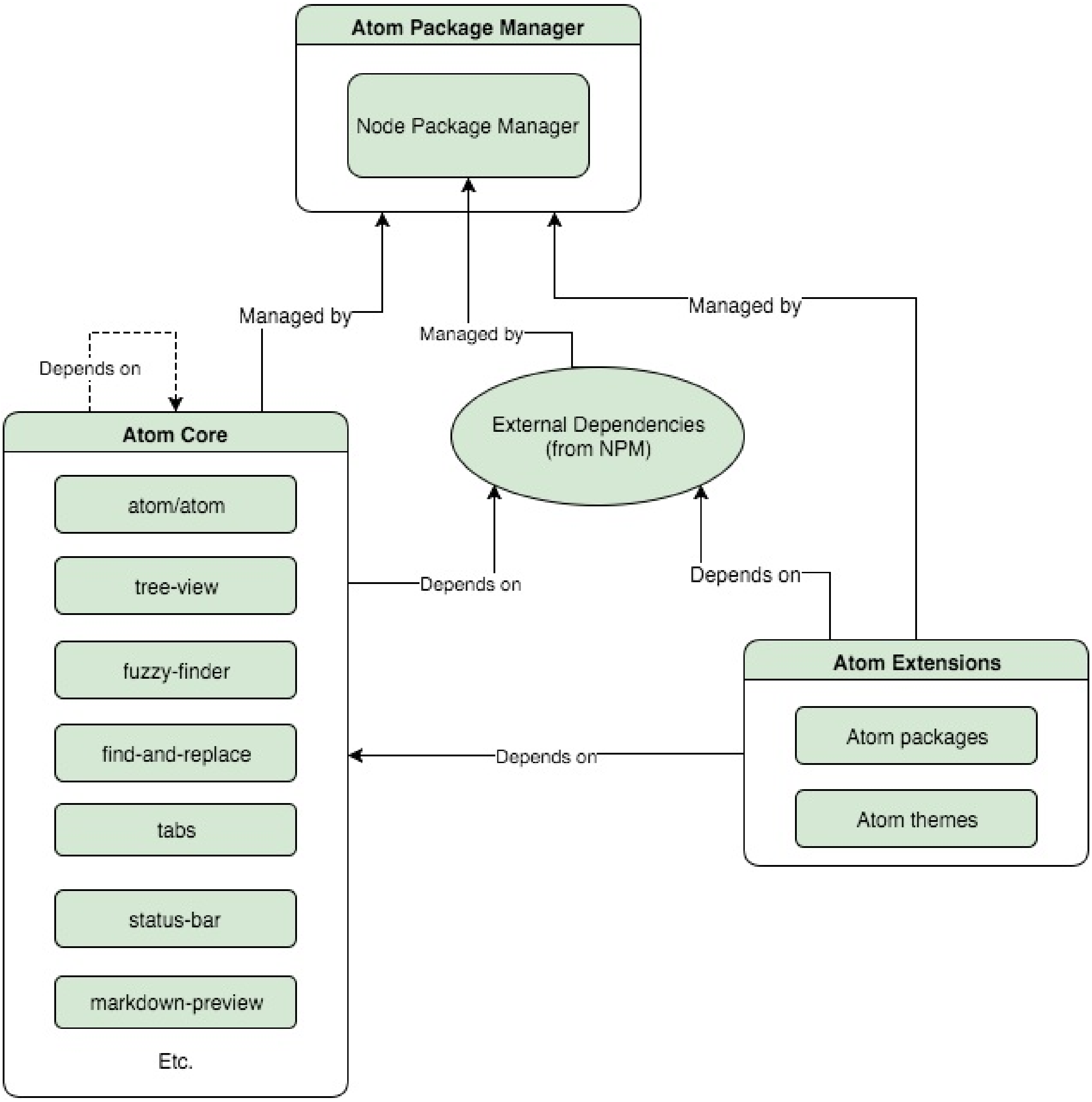
Development View

According to Rozanski and Woods[2], the development view concerns "code structure and dependencies, build and configuration management of deliverables, systemwide design constraints, and system-wide standards to ensure technical integrity". As stated in the Atom Flight Manual, Atom is developed as a very approachable and hackable text editor. This combination raises high standards for the code structure and standardization, as the code needs to be understandable enough to be easily hackable, but well structured enough to be approachable.

> Our goal is a zero-compromise combination of hackability and usability: an editor that will be welcoming to an elementary school student on their first day learning to code, but also a tool they won’t outgrow as they develop into seasoned hackers.

Overview

Atom achieves its goal of hackability by being very modular in its set-up. It consists of more than 4000 packages and themes, which can be divided in two categories: Core packages, about 90 packages and themes that are bundled with Atom and Community packages, extensions developed by the Atom community.



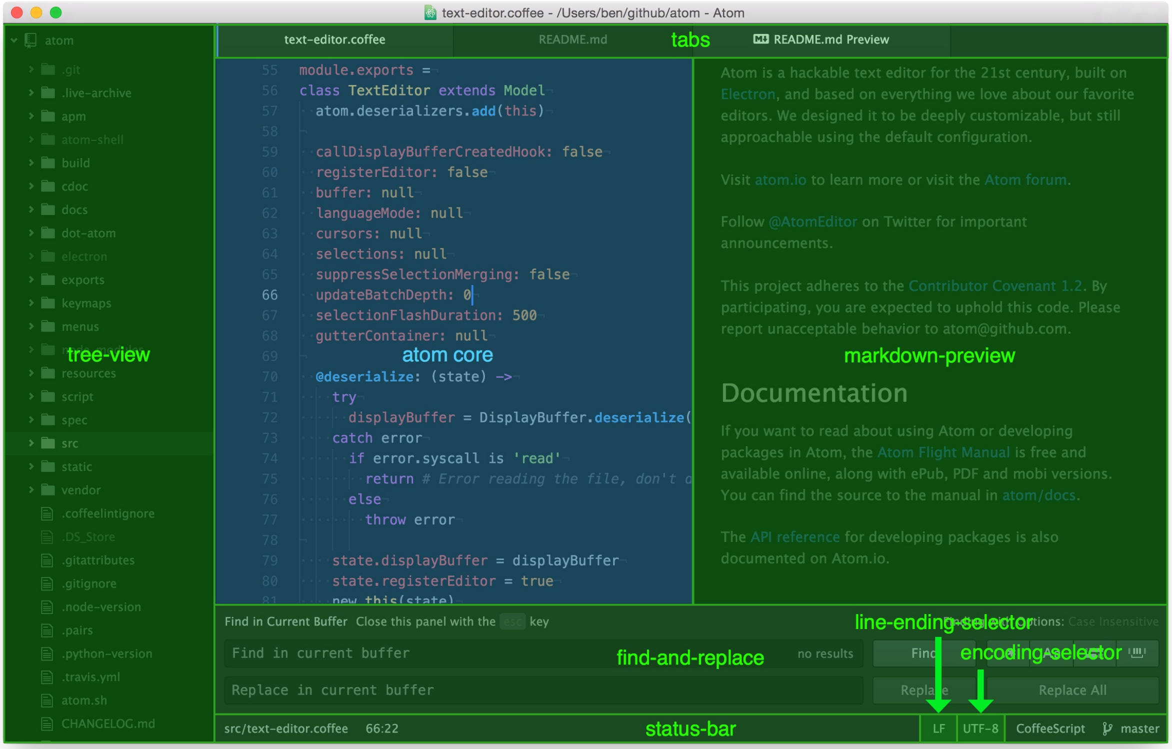
Development View

This image gives a very high level approach to the dependency organisation of Atom. It has been very simplified, as with this large amount of packages it is intractable to display every dependency present in Atom.

The Atom core package contains the most basic editor functionality. The contribution guide of Atom mentions:

> The core editor component is responsible for basic text editing (e.g. cursors, selections, scrolling), text indentation, wrapping, and folding, text rendering, editor rendering, file system operations (e.g. saving), and installation and auto-updating.

All other functionality present in Atom, even one might consider as the most basic for a text editor, is achieved with packages, every functionality typically belonging to its own package. This is represented in the following image, where the core package is shaded blue, and other functionality is shaded green and tagged with the package name.



```
module.exports =
class TextEditor extends Model
  atom.deserializers.add(this)

  callDisplayBufferCreatedHook: false
  registerEditor: false
  buffer: null
  languageMode: null
  cursors: null
  selections: null
  suppressSelectionMerging: false
  updateBatchDepth: 0
  selectionFlashDuration: 500
  gutterContainer: null

  @deserialize: (state) ->
    try
      displayBuffer = DisplayBuffer.deserialize(
    catch error
      if error.syscall is 'read'
        return # Error reading the file, don't d
      else
        throw error

    state.displayBuffer = displayBuffer
    state.registerEditor = true
    new this(state)
```

Find in Current Buffer Close this panel with the esc key

Find in current buffer

Replace in current buffer

Finding with Options: Case Insensitive

Atom is a hackable text editor for the 21st century, built on Electron, and based on everything we love about our favorite editors. We designed it to be deeply customizable, but still approachable using the default configuration.

Visit atom.io to learn more or visit the Atom forum.

Follow @AtomEditor on Twitter for important announcements.

This project adheres to the Contributor Covenant 1.2. By participating, you are expected to uphold this code. Please report unacceptable behavior to atom@github.com.

Documentation

If you want to read about using Atom or developing packages in Atom, the Atom Flight Manual is free and available online, along with ePub, PDF and mobi versions. You can find the source to the manual in atom/docs.

The API reference for developing packages is also documented on Atom.io.

Core package overview[4]

This approach enables a highly flexible developing approach, where large refactorings of a single functionality will not impact other functionality. Atom also enables packages to depend on other packages, replacing the need for redundant work with a simple dependency statement.

Standardization

As an open source project that welcomes contributors, Atom has a great need for standardization. Without it, contributions might not be developed or checked properly, leading to a messy codebase and bugs.

One way Atom fulfills this need is by the contribution guide. This is a document setting a large amount of guidelines for every type of contribution, from bug reports to code contributions to commit messages, along with a clear explanation and pointers to more information.

In addition, Atom is tested using the Jasmine testing framework, which is required for all new functionality. In order to improve this process, Atom uses Continuous Integration tools to do a test build for each new pull request, including running the entire test suite.

Furthermore, pull requests are in almost all cases merged by someone other than the pull request initiator, which will ensure that some other set of eyes checked the code before it is merged.

Codeline organization

The approach of using packages to build up the editor automatically has as a consequence that the code is divided over different packages with dependencies. Every package has a clear description and name, and all the files in a package are specifically for that package.

The dependencies and versions of Atom are managed by the Atom Package Manager, which is based on the Node Package Manager. The Atom Package Manager is therefore fully compatible with the Node Package Manager, and packages from the Node package Manager can be included in the Atom project. To make sure that all the dependencies are using the latest software Atom uses David. David shows outdated dependencies and when possible also the difference between versions in packages.

The Atom core itself consists of about 130 source code files, not counting build & configuration files, styles and tests. Only 10 of these are not in `src` folder, giving Atom a very flat architecture. While the filenames are quite clear, there are multiple files starting with `git-`, `gutter-`, `panel-`, `pane-`, `menu-`, all in the same directory. This suggests the structure could be improved by splitting these into folders. Lee Dohm disagrees:

> They're not really grouped by having the same prefix, they're grouped because that's what the class name is inside them. For example: `panel.coffee` -> `class Panel` ... `panel-container.coffee` -> `class PanelContainer`.
>
> When you get used to using the fuzzy-finder to open files, it doesn't really matter what directory things are in. Also, let's say we wanted to rename `Panel.*` to `Foozle.*`. Right now, all we have to do is change the names of the classes and rename the files. If we had them in a `panel` subdirectory and `panel*` filenames, then we need to change the name of the subdirectory too and that leads to all kinds of clicking around to find things. Having worked on Java projects where it is customary to have deeply nested namespaces where each directory is a namespace component, it makes refactoring tons harder.

The Atom core is tested by about 80 files containing test code, not including the large amount of fixtures used as test cases. These files with test code are again almost entirely bundled into one folder named `spec`.

Technical debt

Technical debt is an important property to examine in every software system, as a large amount of technical debt can make the software resistant to change until it is resolved. The debt is mostly introduced by 'dirty fixes': quick solutions that do not adhere to the coding styles, are not tested properly or are not documented properly.

The definition of technical debt is analogous to monetary debt: you 'borrow' time to develop the dirty fix, but you will have to repay that at a later date, with interest [5]. For example, when a new change is not tested properly, the action of testing this change becomes debt. When this debt is not repaid quickly, the knowledge may be lost, the change will break unexpectedly or other changes will be built on top of it, all increasing developer time needed, representing interest paid over the debt.

A system with a large technical debt is hard to properly maintain, because for there is an outstanding debt to be paid with each change. A common sight in such systems is a simple change requiring a lot of refactoring work.

Current technical debt

Technical debt reveals itself when simple bug fixes and feature requests take a long time to implement, because large refactoring is needed before it is even possible to introduce the fix or the new feature. As such, the issues are the first place to look when assessing the technical debt of the system.

The Atom repository currently has 1700 open issues and even 4282 open issues across the entire Atom organization. The oldest issues are over 3 years old (for reference, Atom was only released to the public 2 years ago), which might indicate a large amount of technical debt. A closer inspection of these old issues shows this is not the case: the developers simply have given priority to other issues or the issue has disappeared from the radar.

A signal of technical debt can be found in the performance issues Atom is experiencing. The Atom repository currently has 51 open issues with the `performance` label, some reported as early as 2013. Additionally, a quick estimation gives that about three new issues about performance are created each week (which are quickly closed as duplicates of the existing issues), indicating the problem is still present. The Atom organization has indicated multiple times (for example here and here) performance is a big issue, not shying away from allocating extra resources. Despite those efforts, the performance issues still remain. This suggests there is a lot of technical debt to repay before being able to fix the performance issues. However, it has also been suggested that the performance issues remain simply because it is a hard problem to solve, especially on the web-based architecture of Electron. Which of these two issues cause the main performance issues can not be determined conclusively.

Dealing with technical debt

The Atom developers have many ways of dealing with, and most importantly avoiding, technical debt. In this section a quantitative and a qualitative approach will be taken to examine this.

Quantitative approach

Atom's contribution guide suggests adding emoji to a commit message in order to make it clearer what a certain commit is supposed to do, opposed to simply adding more features. Examples include 🎨 for improving the structure of the code, 📝 for adding documentation and ⬆️ for upgrading dependencies.

For this quantitative approach, all commit messages from commits made between 1 February 2015 and 1 March 2016 will be examined. This point has been chosen as such to give a large sample of commits in which the emoji guide did not change. In this period 6613 commits have been made containing a total of 2215 emoji's. The following table lists some of those.

Emoji	#	Usage
🎨 (+ 💄)	200	improving the format/structure of the code
🐎	62	improving performance
📝	150	writing docs
🔥	131	removing code or files
💚	60	fixing the CI build
⬆️	1409	upgrading dependencies

The most notable emoji is ⬆️, being used a total of 1409 times. When any of the Atom packages Atom depends on is updated, the maintainers make sure to immediately update the package in the core, to avoid technical debt caused by outdated dependencies. Additionally, they use seperate tooling (David) to keep an eye on all external dependencies.

Qualitative approach

A sampling of issues and pull requests reveals that the Atom maintainers keep a close eye on technical debt and try to avoid it. Some notable examples will be shown in this section.

The maintainers explicitly refrain from adding 'hacks' to code to temporarily fix a problem they believe lies elsewhere. Also for important performance issues, they prefer to fix it properly, instead of relying on a hack to get it done.

The Atom team themselves use the term technical debt as something that should be avoided and actively discuss ways to avoid it. They explicitly compare short term and long term solutions where they juxtapose the added speed of a simple hack to the resulting technical debt it will cause. When technical debt is spotted, the Atom team does not shy away from large refactorings in order to eliminate it.

Additionally, when new features are introduced or even when changing parts of the public API that may break something in another package the team makes sure the changes are properly tested before merging.

When we asked Lee Dohm, Atom Community Manager at GitHub, about technical debt he stated he considered the large backlog of issues as a sign of popularity of the project, and not of technical debt (he added that they are working on tools to better manage the backlog). In addition, Lee stated that a good process is in place for avoiding technical debt:

> We have smart people that communicate well. Anything that requires redesigning, we involve many people in things and go back and forth on it until we come up with a good solution. Even simplistic code is generally well-factored, which makes it easier to change later

In conclusion, the Atom project is very aware of technical debt and actively deals with it. An inspection of the repositories linked to Atom did not reveal many signals of technical debt, indicating that the source code is in good health.

Variability

Atom has a vast amount of features which can extend the basic functionality of the Editor. In the next sections we will give an overview of these features and to explain the relationships between those features.

Atom's strategy on features

As Atom states on atom.io:

> Atom is a text editor that's modern, approachable, yet hackable to the core—a tool you can customize to do anything but also use productively without ever touching a config file.

To achieve this various ways of implementing variability can be seen in the feature model. The most accessible variability are changing settings in the settings menu and downloading packages or themes. Users do not need knowledge about the underlying system to change these.

More advanced users can change some more advanced settings, like altering key bindings or customizing themes in the user stylesheet. Most software developers can alter these files without help.

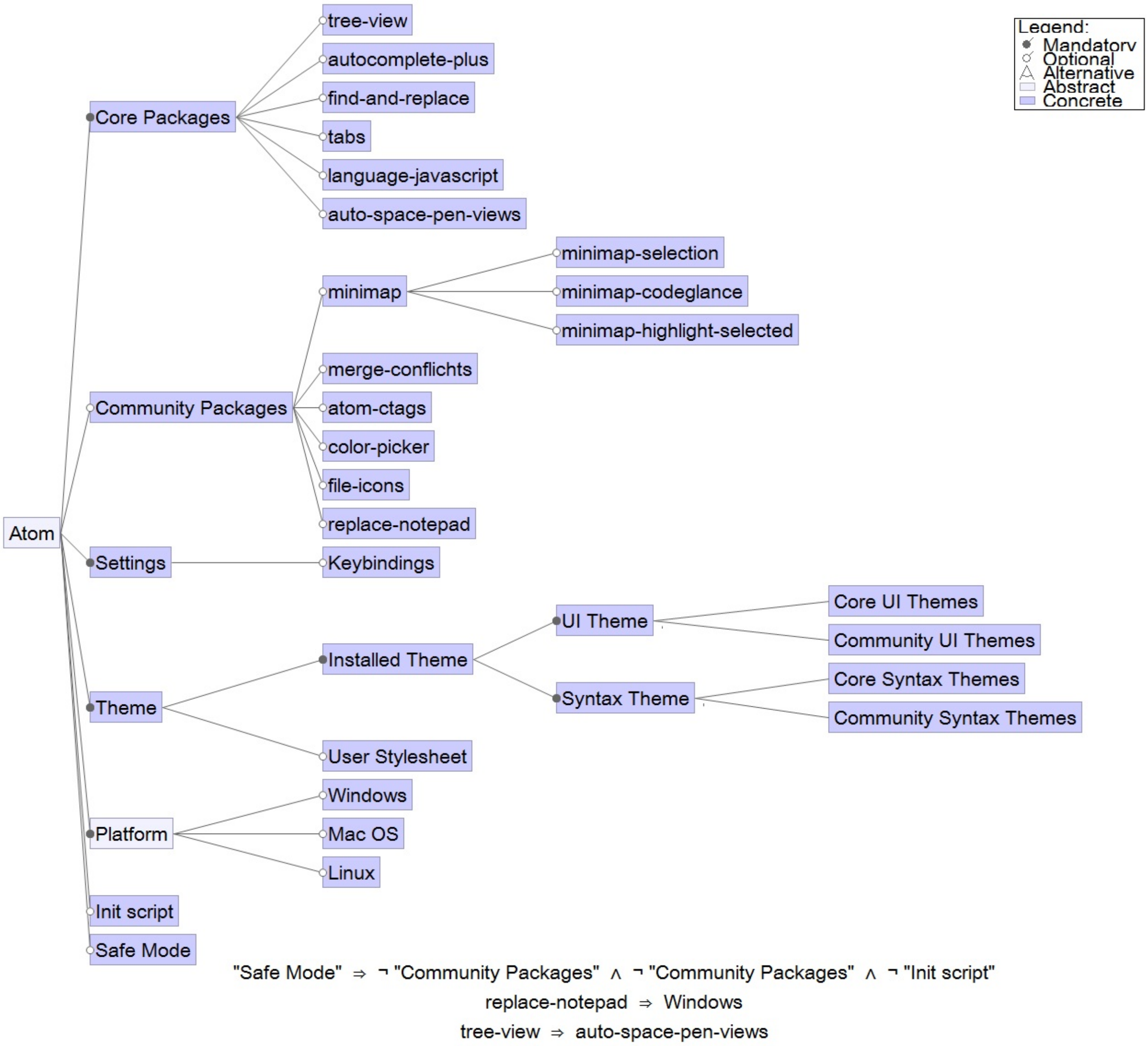
If a user wants to go further than this, he can create his own packages or change the init file. To make this level of customization approachable there is an extensive explanation with example in the flight manual.

Features

Atom has an extreme amount of variability and thus also a lot of features. To put this into perspective Atom already has 3727 packages and 1120 themes at this moment of writing. But Atom not only has these packages and themes as possible variability features, there are for example also settings, snippets, init script settings, etc.

Feature Overview

In this section, we will give an overview of the variability features of Atom. A model of these features, created in FeatureIDE, is seen in the following Figure:



Feature model of Atom

- Most of Atom's default functionality comes with the 77 Core Packages. These are packages bundled with Atom and each contain a specific part of the functionality, such as the find and replace functionality or JavaScript language support.
- Atom can be further extended by installing additional Community Packages. These contain additional features not present with the default installation, for example adding a code minimap or add different icons for different types of files in the tree-view. Packages can also be extended by other packages, for example by adding highlighting to selected rows in the code minimap.
- While every package can have its own settings (not shown in the model for simplicity), Atom itself also has its own settings. These are individual toggles or input fields to change small amounts of functionality. Additionally, all keybindings can be modified.
- The look and feel of Atom is also fully customizable through themes. Atom is bundled with a set of 12 themes and users can install a theme from the community, as with packages. The themes are split into two types, UI themes and Syntax themes. Additionally, users can customize everything using a custom stylesheet.
- Atom is supported for Windows, Mac OS and Linux.
- Users can specify custom functionality to run at start-up using the init script.
- Atom can be started in Safe Mode, disabling user installed functionality for the purpose of debugging.

Core packages

The Core packages of Atom are by default bundled with the Atom editor. These packages create the main functionalities of the Atom editor.

Atom has many of these core packages, for example tree-view which adds the overview of files in the project directory. But also packages like autocomplete+ are part of the core and adds the autocompletion for different languages while typing.

Automatic updates are done in run time without need for local compilation. Only when a developer changes the source code locally it needs to be recompiled.

Community packages

The community packages are extensions which are developed mainly by the Atom community. They can extend the default Atom functionality in many ways.

For example a package adding a minimap with the outline of a document. Which can then again be extended by minimap-selection which adds features that show the selected text in this minimap. But there are also packages like merge-conflicts which help you resolve git merge conflicts inside the editor.

Like the core packages, community packages bind at run time except for local source code changes. To ensure packages work immediately after installing them Atom uses lazy loading. In contrast to core packages, updates for community packages need to be confirmed by the user.

Settings

Although some features can be enabled by downloading extra packages, some features are properties of the core packages. These features can be altered in the settings menu. Some examples are: adjustable line height, key bindings, non-visible character handling. All changes in the settings are bound at run time.

Themes

Besides packages, themes can be installed as well. These themes are installed the same way as packages. A theme can change the look of the editors UI, but also the syntax highlighting. While there are a lot of themes available, these themes do not have any dependencies. While some of the themes look better with certain languages or have specific stylings for packages, they do not require these languages or packages.

Themes can be divided into two categories: Core themes and Community themes. While core themes are bundled with Atom and can be chosen from the settings directly, the community themes must first be installed trough the Atom package Manager. After installation themes work instantly.

Platforms

Atom is currently available on Windows, Mac OS and Linux. Atom has always been available for Mac OS from the beginning. In July 2014, Windows support was announced. This supported was extended with a proper installer in December 2014. In November 2014, Linux support was announced. Nowadays, every feature is supposed to be working on every platform, as evidenced by the Contributing guide explicitly mentioning to avoid platform specific code and continuous integration tools running on all three platforms.

Because of the underlying Electron architecture, there are very few differences in the functionality on the different platforms. Only some small differences exist, because some features make more sense on some platforms than others. For example, the items in the menu bar are divided differently on different platforms because the different platforms have different standard practices for the menu bar. This sometimes results in small issues, such as the one fixed in our first pull request.

In addition, some community packages work only on specific platforms. This largely is because the functionality itself is specific to a certain platform, such as the replace-notepad package which tweaks the Windows registry to replace Notepad, or the spotify-linux package which speaks to Linux's Spotify client.

Init script

The Atom editor has a special initialization script called `init.coffee` which can be opened trough the Atom menu. This initialization script is run when the Atom editor is opened. It enables the functionality to run specific CoffeeScript code to further extend the functionalities of the Atom editor.

To give an example where the init script could be used:

```
atom.workspace.observeTextEditors (editor) ->
  editor.onDidSave ->
    console.log "Saved! #{editor.getPath()}"
```

This CoffeeScript code adds a logging message in the development console whenever a file is saved. This enables a developer to better debug a problem with the saving of files. Changes in the init script will bind on the next startup of Atom.

Safe Mode

Atom can be started in Safe Mode with the `atom --safe` command. When started in this mode, Atom disables three things: all community packages, all community themes and the user's `init.coffee`. Safe Mode is mostly used for debugging purposes, as it provides an as clean as possible environment to reproduce behavior.

Relationships between features

The core of Atom is that most features are independent which greatly improves the extensibility and reliability of Atom. Still there are some features with dependencies, but this is mostly because these features extend the functionality of the features which they depend on.

For example tree-view depends on atom-space-pen-views, which enables the functionality to create views. These views are needed to show the tree-view in the GUI.

But also many packages depend on coffeelint during development, which adds the ability to check coding style. This makes sure that the coding style across different packages is clean and consistent.

Functional View

Atom has a lot of functionalities build which are available in the editor. For the functional view of Atom we first look into the Capabilities Atom has. Then we will show what interfaces Atom uses and finally we will show how this is structured.

Capabilities

Every package in Atom adds functionalities to the Atom editor. To analyze the functional capabilities of Atom we look into the most important packages and their functions.

Package	Function
core	Package API and main text editor
tree-view	Folder structure overview from working directory
tabs	Multiple text views in tabs to switch between files
notifications	This shows messages and error boxes
settings-view	Shows settings and makes them editable in a view
autosave	Makes sure files are saved in the background
find-and-replace	Find and replacing text in a file
status-bar	Shows status of the current file and cursor

Next to these above mentioned core functionalities there are a lot of functionalities available which are optional. Some of these functionalities are also included into the Atom core but most of them are available as Community packages.

External interfaces

Atom has two main interfaces, the interface for the user to interact with, and the interface for packages to interact with the Atom core.

User-interface

The user-interface is a Graphical User Interface, or GUI. This interface has all core elements that exist in the core package, this is what makes sure that Atom is consistent in the GUI. But the user-interface can be extended with different packages, that add functionality and user-interface elements.

Package-interface

When developing packages for Atom, communication between the Atom core and the package that is being developed, is needed. This is done via an API. This API is the package-interface that can be used to use functionalities that already exist in the Atom core. With this come a lot of functionalities. A few of these functionalities are for example:

- `Notification` and `NotificationManager` provide the functionality to create and manage notifications that need to be shown to the user.
- `TooltipManager` manages the tooltips that can be shown to the user.
- `Emitter` handles all the events happening in the code. It is not only for emitting events, as the name suggests, but also for handling them.

Internal structure

The capabilities of Atom are realized by coffeescript files in the src folder. The src folder of each package contains all functionalities for the editor. When building a package these coffeescript files are compiled into javascript files.

The API is created automatically when building. From the user-interface the functionalities of the API can be called by using shortcuts as defined in the `keymap` , using the command as defined in `register-default-commands` or using the visual interface. For the visual interface the bootstrap environment offers `click` , `hover` and `focus` options that can be used and linked to actions by packages.

Conclusion

This chapter has shown that Atom is a text editor with a lot of functionalities. Because of this extreme amount of functionalities in Atom, they divided the code into separate packages. This enables users to easily extend the functionalities of the Atom text editor. Besides these functionalities we also discussed the involved stakeholders and analyzed the context of Atom. We also looked at the technical-debt of the Atom project, which they seem to deal with very well. After all these extensive analysis of the Atom project we come to the conclusion that Atom really holds up is promise of a hackable text editor.

References

1. http://blog.atom.io/2015/06/25/atom-1-0.html. *Atom 1.0*. @benogle (2015).
2. Rozanski, N., & Woods, E. (2012). *Software systems architecture: working with stakeholders using viewpoints and perspectives*. Addison-Wesley.
3. https://atom.io/
4. https://github.com/atom/atom/blob/master/CONTRIBUTING.md. *Contributing*.
5. http://martinfowler.com/bliki/TechnicalDebt.html
6. https://github.com/facebook/nuclide/graphs/contributors

Appendix: Contributions

During the analysis of the architecture of Atom, we additionally made some contributions to Atom.

Pull requests

1. **Change menu names for different platforms** fixes the text in the welcome screen of Atom for different platforms.
2. **Add zero to hexadecimal numbers below F (16)** fixes the problem with the color picker generating incorrect hexadecimal numbers.
3. **Add defaults on focus** adds the functionality that defaults of settings are entered on focusing the input field.
4. **Add summary chapter 3** adds missing documentation to the Atom Flight Manual.

Others

1. **Atom hangs for ~30s when pasting a large text block into a new tab** is an issue where we were able to add more information by reproducing and profiling the error.
2. **Uncaught RangeError: Maximum call stack size exceeded** is an issue created by us which we found by trying to reproduce the above issue.
3. **editor:select-to-end-of-line should work similarly to editor:move-to-end-of-line** is an issue for which we did some initial development.

BigBlueButton: High-quality on-line education

By Michail Anastasakis, Pietro Frigo, Manuel Valle Torre and Kristín Fjóla Tómasdóttir
Delft University of Technology

Abstract

BigBlueButton is an open source web conferencing system for on-line learning. The system has been in development for nearly 10 years now with a strong group of core developers with a special focus on the stability and modularity of the system. For people interested in software architecture, it is insightful to study the structure of such a well established software project like BigBlueButton. Hereafter, a number of views and perspectives are used to demonstrate important aspects of BigBlueButton's internal and external structure. This chapter provides a description of the people and systems involved, the developing process and architecture, the various features of the software along with its variability and possibilities to evolve.

Table of Contents

What is BigBlueButton?

Before starting the itinerary throughout the whole architecture of BigBlueButton (BBB) it is useful to understand what the system does. So, let's begin.

Quoting their website:

> BigBlueButton is an open source web conferencing system for on-line learning.

The project originally started at the Carleton University in a program for innovation and entrepreneurship in engineering [5]. The program was already using a commercial conferencing system for their international students but it was too expensive. A student of the innovation program, Richard Alam, received a challenge from the director of the program to replace the current system. Today, BigBlueButton is a well established software which is widely used and integrated into many popular systems [6] [7]. Their vision is that every student should have access to a high-quality online education.

BBB has traditional conference system features such as a chat and webcam sharing for users to be able to interact in a meeting, and a deskshare functionality for the presenter to share his desktop with the audience. However, the drive of BBB is education so there are two main characters in that environment: *teachers* and *students*. To satisfy the modern technology that these two groups require, features such as an interactive whiteboard is also present to aid the learning process. Another cornerstone is the recording system that allows the students to watch the recordings whenever they prefer, removing the time constraints imposed by normal lectures. These functionalities, amongst others, will be analyzed more in detail in later sections, along with the structure around their implementations.

Who cares about BigBlueButton?

A software is made by people for people. There is usually a group of people contributing to the software project and taking interest in its development. The software itself is also made to meet the requirements of some other, or even the same, group of people. These people who are somehow involved in the software or take interest in it are called stakeholders. It is therefore very important to identify them in order to understand how the software should be implemented.

Blindside Networks is by far the most evident and important stakeholder of BigBlueButton. It is the company that started the BBB project and provides hosting for it along with commercial support. The company is crucial for the project since no revenue comes from selling the product (since it is open source) but only from providing hosting and support [16]. Based on the types of stakeholders that Rozanski and Woods suggest [11], Blindside Networks can be considered as *Acquierers*, *Assessors*, *Communicators*, *Developers*, *Maintainers*, *Support staff* and *Testers*. Its two founders are Richard Alam and Fred Dixon, which are the most influential people in the system. Alam was the first developer of the project and is still today the lead architect along with being the CTO of Blindside Networks [5][25]. Dixon joined him after the first release was made public and is the project manager of BigBlueButton and the CEO of Blindside Networks. They are both still active developers and manage all aspects of the development.

There are other important groups of stakeholders that can be categorized in the following types:

- **Developers**
  Contributing to the BigBlueButton project on GitHub there are 61 developers which includes a number of about 20 very active members [26]. The major commiters are again Richard Alam (3691 commits) and Fred Dixon (694 commits). Other important developers, especially for the 1.0-beta release, are Tiago Jacobs from iMDT who develops for the Red5 media server which is used by BigBlueButton, Felipe Cecagno working for MConf on the client, Chad Pilkey who contributes from the Seneca College working also on the client and Calvin Walton who is a part of Blindside Networks [27].

- **Communicators**
  The role of communicators is to explain the system to other stakeholders. For BigBlueButton these are the people responsible of writing the documentation, writing blog posts and those that are taking care of answering questions on the mailing lists available for users, developers and setup. Blindside Networks put a lot of effort into making a good community around BBB, especially by upholding the mailing lists [16]. All three forums are very active with many people offering their help, but most commonly it is Fred Dixon that answers the forum posts and sometimes the hosting company HostBBB which mainly answer questions about its service. Another communication

method is the BigBlueButton summit which is held once or twice per year since 2012 where developers gather to discuss further improvements and workshops are conducted as well.

- **Customers**
  The users of the system have been mentioned before to be teachers and students. The customers of BBB are however not these users but instead the companies that adopt the system. These customers do not pay for the usage of the software since it is open source, but they might pay a company like Blindside Networks for hosting and support. These customers are mostly schools and universities (such as Boston College), but also companies that integrate BBB into their Learning Management System (LMS) and system of e-learing (such as Schoology and Moodle) [7][28]. Another important user that adopted the system in 2015 is the Defense Information Systems Agency (DISA), extending the user community to governmental institutions as well [29].

To identify which stakeholders are necessary to manage closely it can be helpful to place them in a Power/Interest grid as can be seen in Figure 1:

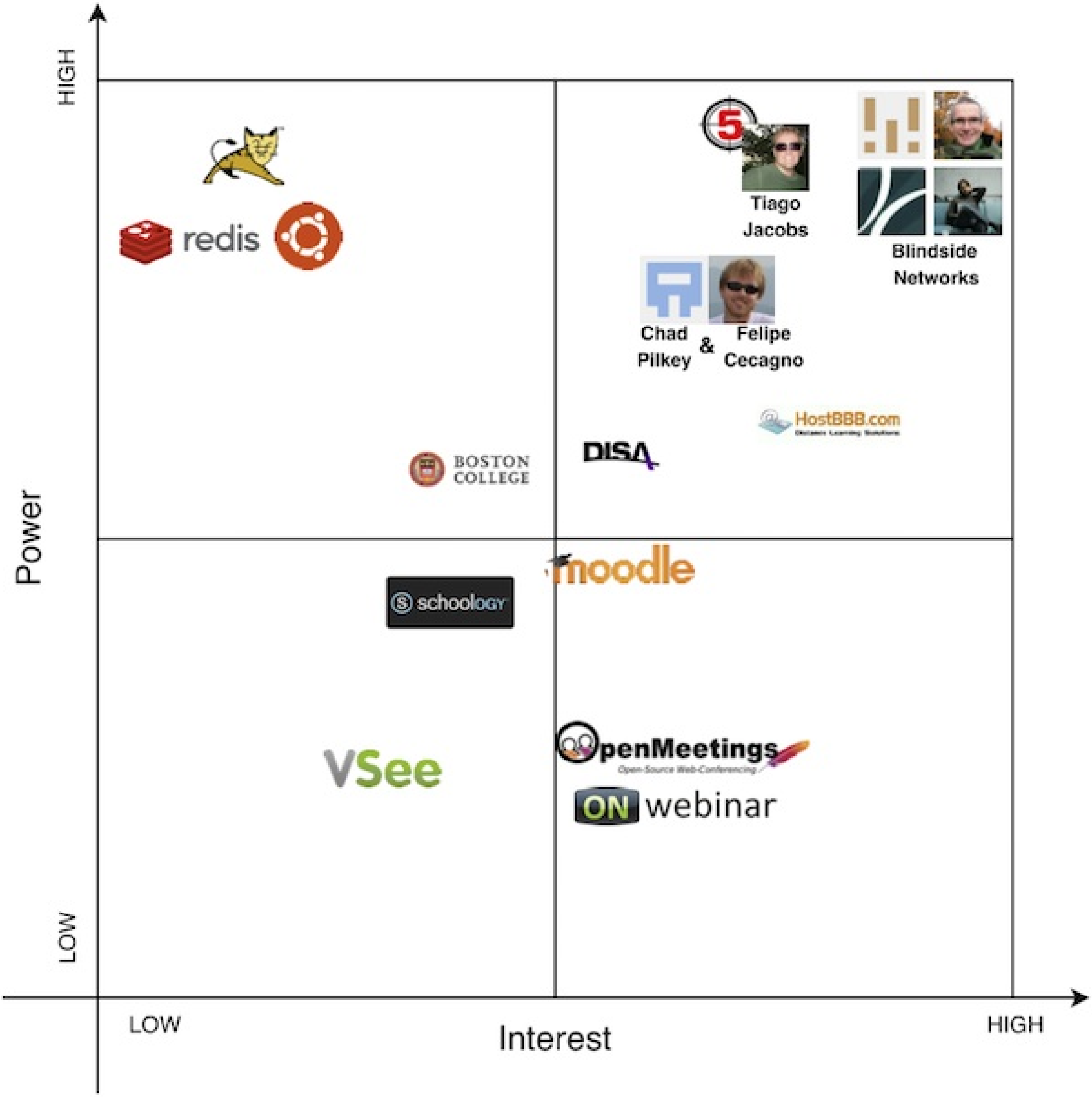


Figure 1: Power/Interest grid of stakeholders

The Power/Interest grid is divided into four blocks based on the level of power and interest [15]. The stakeholders in the lower-left corner should only be monitored, the ones in the lower-right corner kept informed, the top-left corner kept satisfied and the top-right corner should be managed closely. Additional stakeholders are present in the grid which are competitors such as VSee, OpenMeetings and OnWebinar along with some software that the system relies on such as Tomcat and Ubuntu. The most important stakeholders are placed in the top right part of the grid which are in this case Blindside Networks, the core developers and the hosting company HostBBB.

How is it made?

The BigBlueButton team treats the project as a product by having a standardized development process and release cycle along with upholding a solid architecture [16]. Hereafter we will describe their development process, how the software is constructed and what components they use.

Development process and releases

The BigBlueButton group puts a lot of effort in making good quality software, both for users and developers. For the users their focus is mainly on stability and usability, also with an emphasis on features [4]. For developers they try to keep their technical debt low by focusing on modularity and with refactoring in each release. Additionally, they keep up excellent documentation and uphold good code quality. There are certain conventions for the developers to follow regarding the code such as Java Coding Conventions by Oracle, AsDoc format for Flex/ActionScript code and JavaDoc for documenting methods.

Code and issue management is entirely done on GitHub while most discussions occur on the dev mailing list. A typical occurrence is that a BBB developer creates an issue on GitHub for a bug or an improvement and Dixon (being project manager) assigns it to an appropriate developer. Once the issue has been solved, a pull request is made by that developer where he links it to the issue. Finally the pull request content is reviewed and merged by some other developer, often by Dixon or Alam.

Since the first release in June 2009, there have been 14 releases with the latest one in October 2015 (release 1.0-beta). Each release has a well defined development cycle as can be seen in Figure 2.

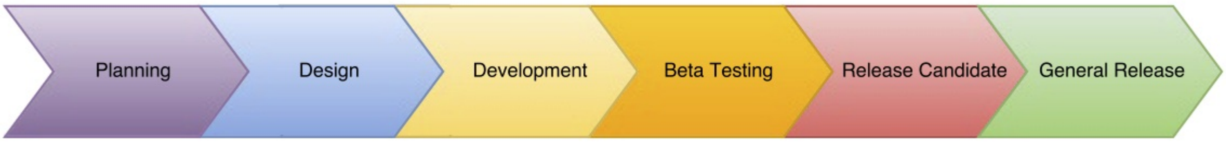


Figure 2: Development process for BBB

1. **Planning**: The development cycle starts with deciding on the main features that the BigBlueButton team wants to implement in that release.
2. **Design**: For each feature, issues related to them are created and specifications are published by the lead developers for the bigger and more complex ones.
3. **Development**: During this phase, developers start working on their own forks and make pull requests (PR) against the master branch. Every commiter is responsible for the PR to be compatible with the master branch but it should always be reviewed by another developer with some experience in the context of the PR. If the experienced reviewer considers the PR to negatively effect the stability or usability of the project he or she will reject it. If the PR gets accepted it will be tested with usage by other developers until a release is published.

4. **Beta Testing**: Before releasing the software is heavily tested by going through at least one beta stage and sometimes tested by the community for months. In this period every bug that is found is fixed.
5. **Release Candidate**: In this step, the label of the build is changed to be a *release candidate* and the core developers wait for more feedback from the community, to fix the last possible bugs.
6. **General Release**: When there are no bugs left and the version is completely stable (this usually means two weeks of no reported bugs) the tag is changed and the *general release* is published.

These steps construct the development process in BigBlueButton and what is interesting to notice is that the release is never planned for a specific date but only requires the system to be stable. As mentioned before, this highlights the focus on stability that the development team exhibits.

The world outside BigBlueButton

It is important to identify how BigBlueButton works with other components in order to develop and maintain the software. The external entities that are connected to BBB are displayed in Figure 3.

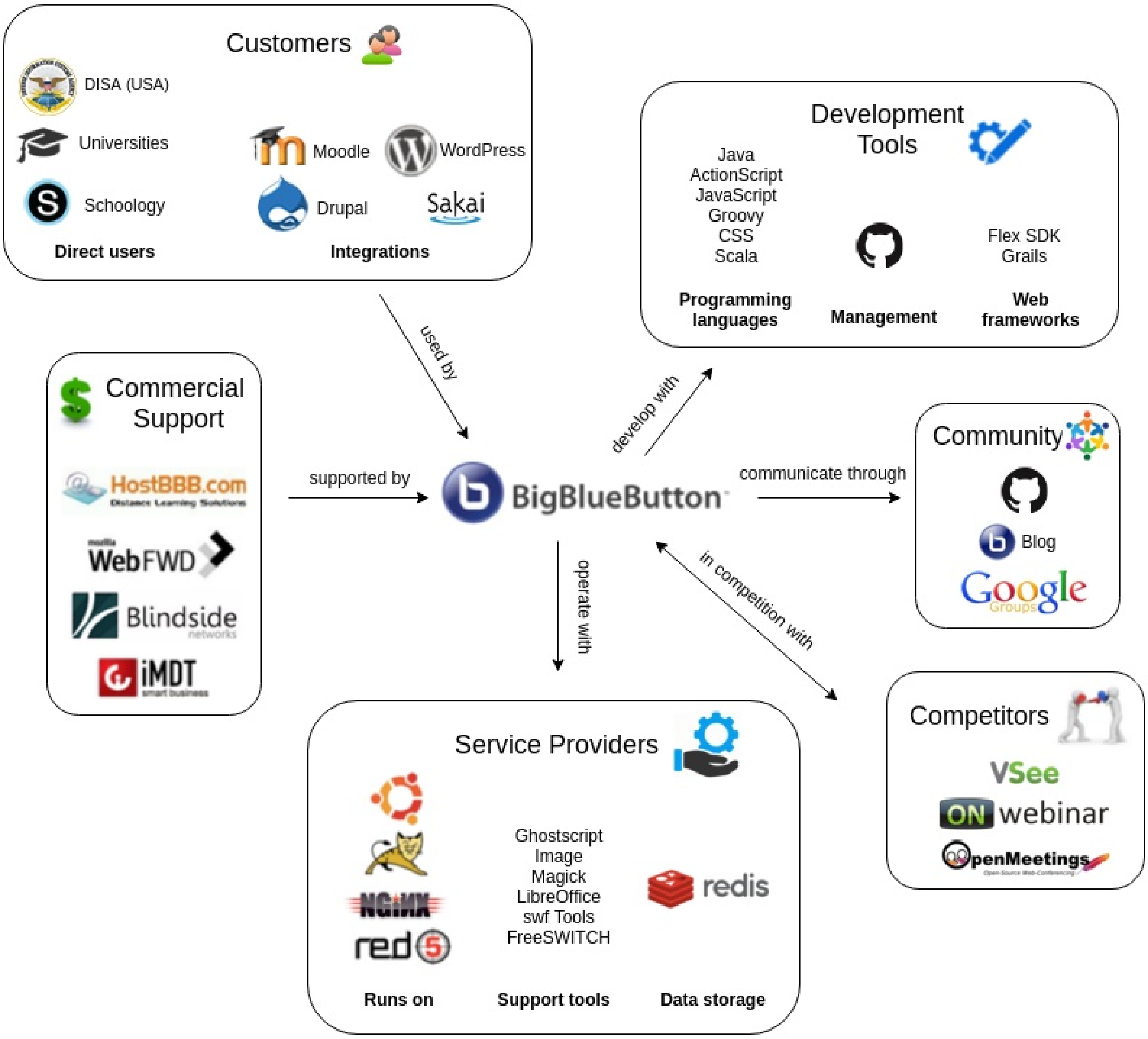


Figure 3: Context view of BBB

BigBlueButton is an open source project and as a result needs help by the community to grow. Developers and users discuss about issues and new features through GitHub and mailing lists (*community*). For developing the software, many languages and frameworks are used such as Java, ActionScript and Grails (*development tools*) [17]. In order to make the system run, BigBlueButton decided to operate through external entities such as Red5, Tomcat and redis who provide their structure and implementation (*service providers*). In order for a project like BigBlueButton to grow it needs financial and operational support. In the past BBB received support from the WebFWD program and nowadays Blindside Networks is the main support provider (*commercial support*) [5]. The customers of BigBlueButton can be separated into two different categories, the direct customers who include universities like Boston College or even the Defense Information Systems Agency and third parties that integrate BigBlueButton into their own system with the most famous being Drupal and WordPress (*customers*) [7]. Finally, as with all software there are competitors to BigBlueButton with some of them being VSee and ONwebinar (*competitors*).

The world inside BigBlueButton

The architecture of the system is split into layers and modules. The main layers in BBB are Client, Server, Database and Configuration. Inside them, the different closed and open source projects interact across the platform as can be seen in Figure 4.

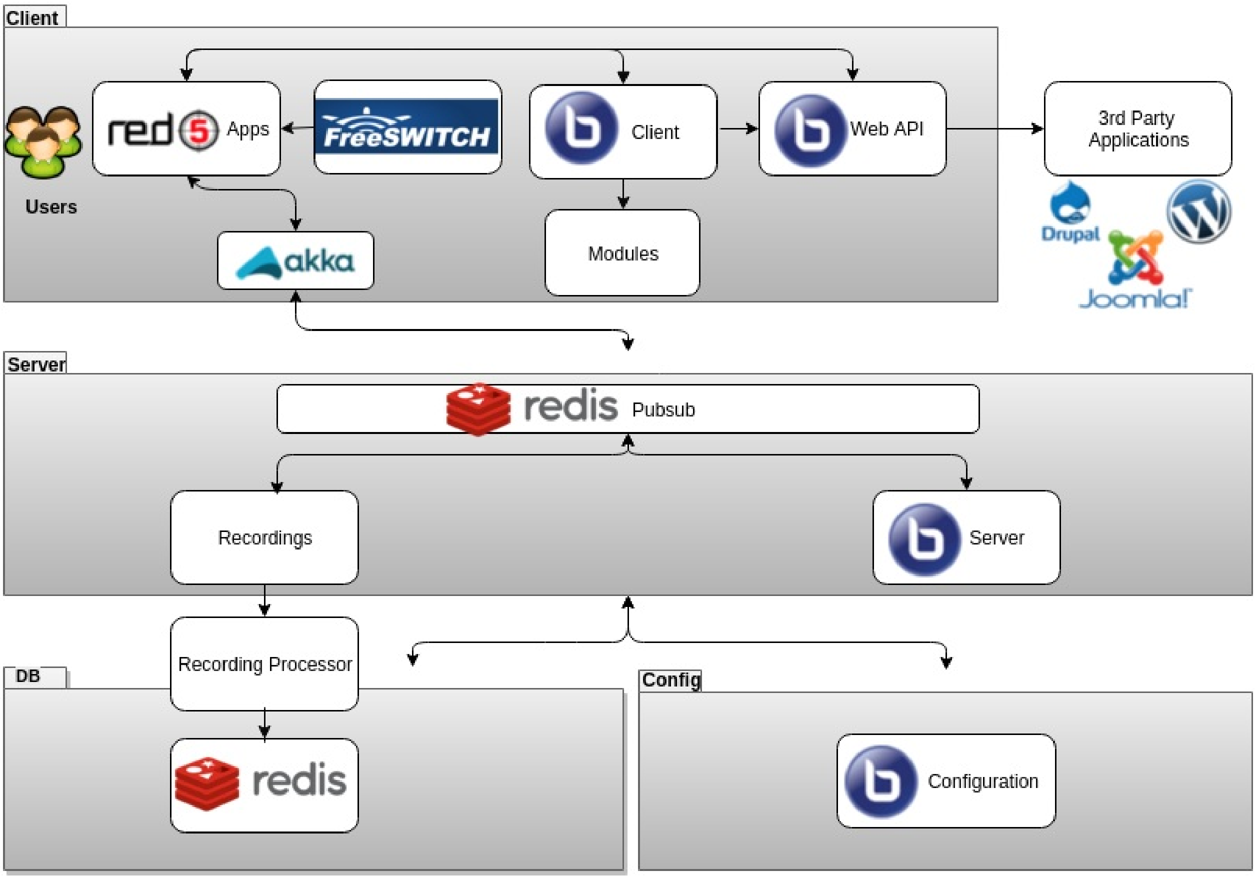


Figure 4: Architecture of BBB

Client Layer: The first layer of BigBlueButton is where the user performs the most activities. We can find the Red5 Apps that provide the functionalities for media sharing and streaming such as chat, board, presentation, voice, video, as well as FreeSWITCH which is the voice capability for conference [18]. This is all connected to the BBB Web API which enables the use of other 3rd party applications such as Drupal and Wordpress. The client itself consists of various modules which serve different features of the software. Finally, there's the Akka apps that put together all the elements on the client side for a smooth real-time experience.

Server Layer: For the next layer, the first component is the Redis Pubsub which acts as a channel between server applications, connecting the main BBB Server and side applications for a unified performance. The HTML5 Server is not depicted since it is, as mentioned before, in process of implementation. The other part of the server level is the Recordings part of conferences, which manages all the events such as slides, videos, webcam and desktop, separately.

Database/Configuration Layer: For the final layer, we have the the Redis Database which is used for saving all the data of a conference session, which is then processed by the Recording Processor. On the configuration part of the layer we have the BBB Client Config file. The `config.xml` file includes the settings of all the modules that integrate with the BigBlueButton application, the logging system configuration and configuration settings concerning the servers IP and permissions.

What does it offer?

In the introduction a few features have been presented only to give a small idea of what BigBlueButton button is. Now it's time to go more in depth. The implementation of these features is divided into the separate modules in the client. These modules will be discussed after describing the various features of the software.

Features

As already said the two main categories of users are *teachers* and *students*. Therefore, it is logical to analyze the functionalities of the system based on these two roles.

Teacher features

The teacher is able to upload his slides so that everyone taking part in the class can see them [23]. He is then aided in his presentation by an interactive whiteboard that shares his pointer and notes on the slides, making it easier for students to follow. The teacher is also able to share his own desktop in order to go beyond the presentation itself or to fix a specific layout to have the students focus on a specific aspect of the lecture. When he wants to make the lectures more interactive he has the possibility to have a student presenting the lecture. This means that the student whiteboard will be shared for that specific timespan. Moreover, he is also able to set up polling sessions or to split the students into breakout rooms for group projects. Finally, for students that are not present in the class, the teacher has the opportunity to choose to record the lecture.

Student features

Students are able to set a reaction to the lecture with emojis next to their name, giving a possibility for the teacher to understand the prevalent reaction of the class to the lecture [24]. These emojis include a raised hand that allow them to ask questions. The students are then able to take notes on the slides so that they can avoid to lose important parts of the lectures and they are even able to share a common notepad between multiple students where everybody can contribute to the notes. In the case that the teacher decided to record the lectures they are even able to watch them back whenever they want.

Shared features

The system has also a few functionalities that are shared among all the users. One is a WebChat on which everyone has the possibility to have private conversation with specific users or publicly interact with all the others taking part in the class [31]. There is also a WebCam and a Voice over IP (VoIP) system that makes the communication more natural.

Modules

The system has been implemented with a modular structure in order to decouple functionalities as much as possible. Modularity is in fact one of the main goals in the development of BigBlueButton [9]. Each module is responsible of a few of the functionalities listed above. In Figure 5 these modules are represented showing the dependencies between each of them.

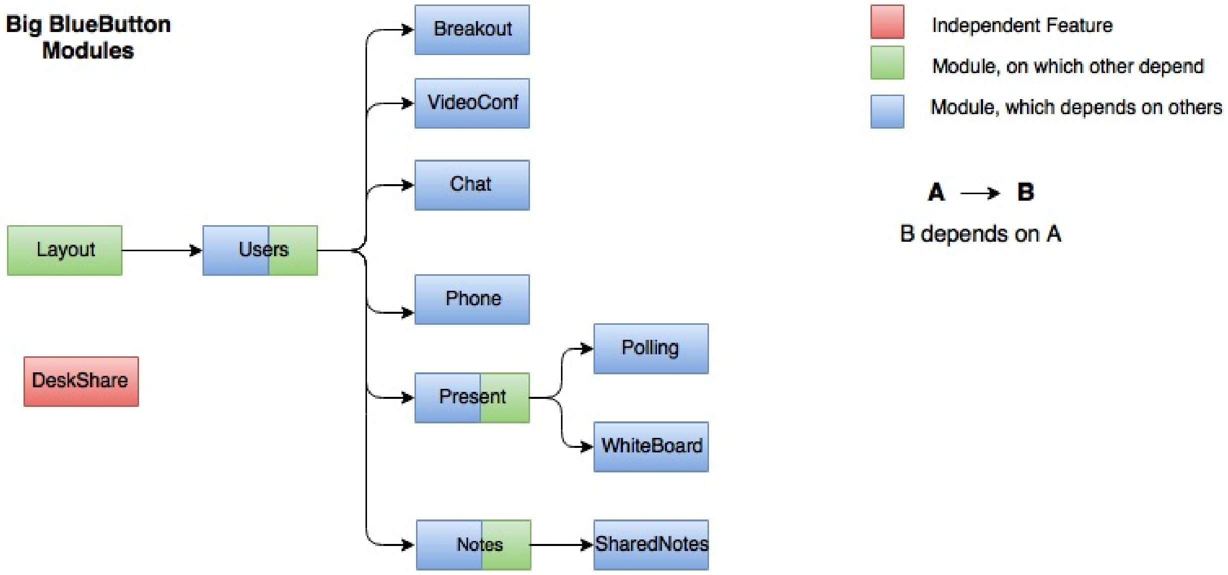


Figure 5: Modules of BBB and their dependencies

There are also two additional modules currently under development [9]. These are a closed captioning module that has the purpose of enhancing the accessibility of the system [21] and a synchronized playback module for external media [22].

Module communication

In order for the modules to work efficiently and coordinated, a shared structure and communication is required. For that reason a specific design pattern is followed when modules are constructed. The communication of modules is implemented by messages being broadcasted and handled by each module for every action in the system. Furthermore, every module has a similar folder structure with each folder being responsible for a different aspect of the module functionalities.

More specifically, every module has a *Service* folder that provides the *Message Receiver* and *Message Sender* which are classes responsible for handling the various event messages broadcasted by the module. Moreover, the *Events* folder includes the specific events related to the module and the *Managers* folder includes the manager who is responsible for the initiation and termination of the module. Finally the *Models* folder includes the different module configurations (Figure 6).

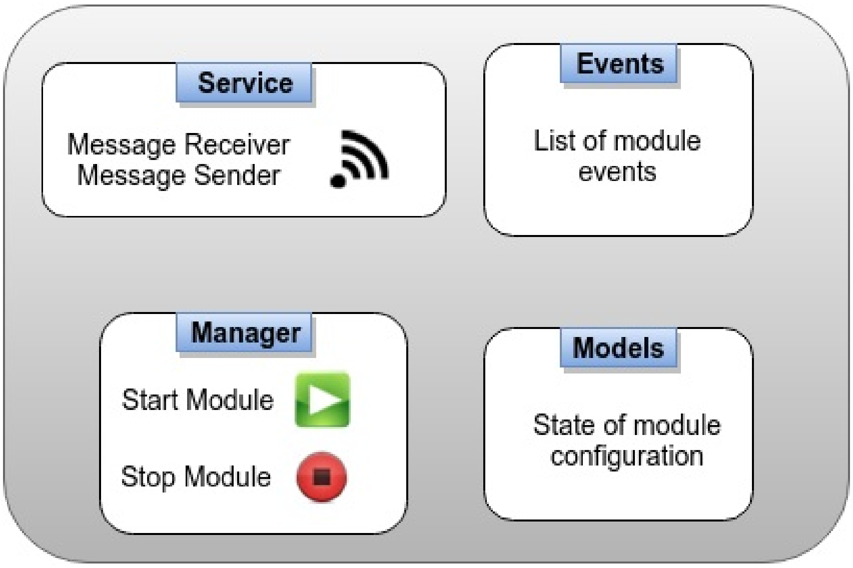


Figure 6: Internal structure of the modules of BBB

Is it implemented well?

For a project to evolve efficiently, it is important to take into consideration future consequences during development. If these consequences are not considered, developers might introduce technical debt which could slow down future development. Like stated before, the developers of BigBlueButton try to keep the technical debt low by enhancing modularity in the system and refactoring for every release. To investigate the technical debt in BBB we applied the SonarQube tool on the project along with some manual analysis.

What does SonarQube say?

SonarQube bases its measures of technical debt on the SQALE methodology [19]. Every time a code breaks a rule an issue is raised and is categorized as a *blocker*, *critical*, *major*, *minor* or just an *info* which is not considered as a flaw but merely a finding. All these issues are converted into a measure for technical debt which is described in time units. We applied the SonarQube tool on the BBB project, analyzing all code written in Java, ActionScript and JavaScript which accounts for 82.3% of the code.

The results from the analysis show a passing grade of A which means that it has low technical debt. The amount of technical debt was reported to be 387 days and 32.534 issues with most of them being *major* or *minor*. Furthermore, looking into the files that have the most technical debt (the highest six files) we see that they are all external files (not written by BBB developers) like `jquery-ui.js`, `jq-ui.js` and `sip.js`. For internal files, instead, the number of issues per file is relatively low and the classes' size is not too big depicting a high quality project. In Figure 7 some of the project's files are visible where the size of the circles reports the technical debt with the number of issues on the y axis and file size on the x axis. All circles outside the dense lower-left corner are classes from external sources.

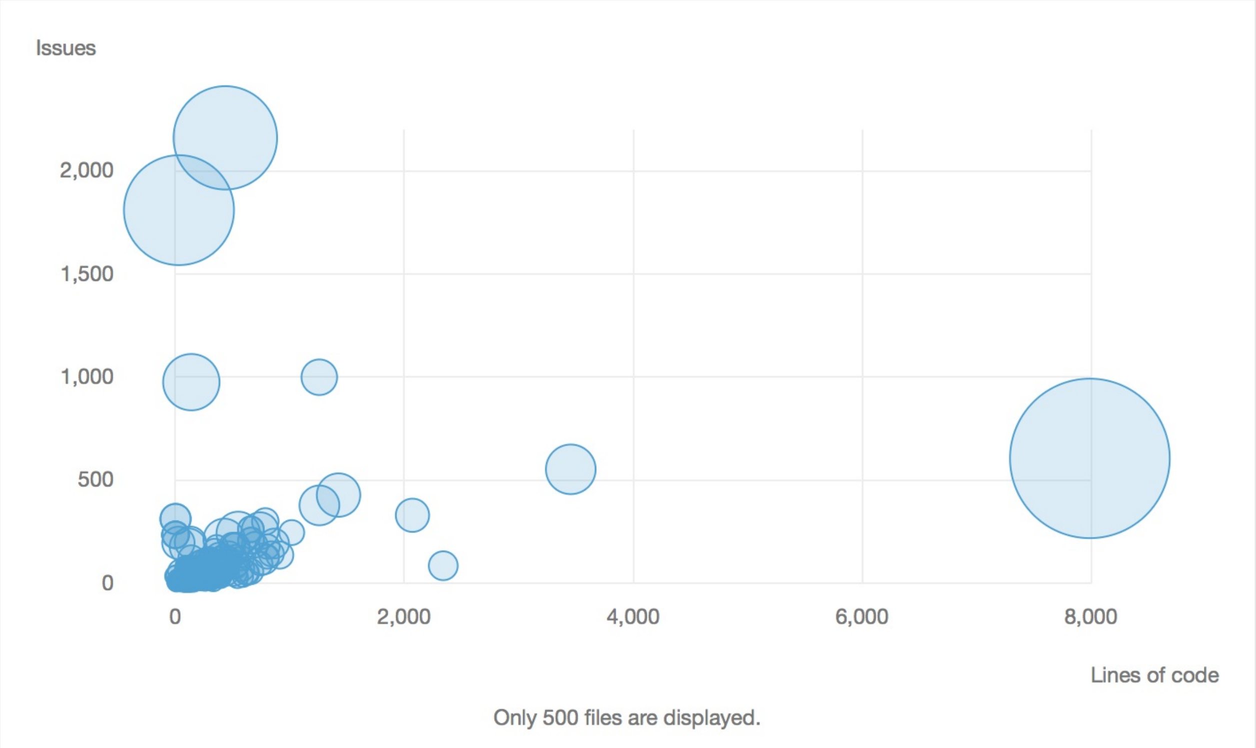


Figure 7: Technical debt of files in BBB

What did we find out?

Based on the work by Cairns and Allen [1] we analyzed the project in regards to its velocity rate, stressful releases, aging libraries, defects and automated test coverage.

1. From the GitHub repository we observe the graph in Figure 8 depicting the number of commits from March 2015 up to February 2016 [30]. It is evident that the velocity rate is decreasing which might be an indicator that too much technical debt has piled up and is now impairing development productivity. It might also be because of other reasons, like less features being added now than before.

2. If the team always experiences stressful releases or if bugs accumulate in the system it might very well indicate technical debt. Like described earlier the BBB team states that they take their time with releases where they make sure that they have fixed every single bug that has been introduced. However, looking at the issues on their GitHub repository (at the time of this being written), out of the 336 open issues there are 155, almost half of them, with the label `Defect` which still have to be fixed.

3. On the aspect of aging libraries there are not any issues since all libraries used are up-to-date.

4. Without being able to analyze test coverage due to the multi-language nature of the project, there seems to be a serious deficit of automated tests. They heavily test the software before a release but that is almost entirely done with usage by the community [18].

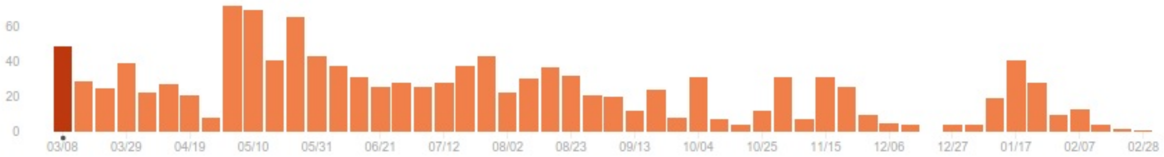


Figure 8: Commit history of BBB 2015-2016

Finally, by looking at the analysis above it is evident that the BBB developers try to keep the technical debt low and that they succeed in doing so. By doing that they minimize the development time for each release since technical debt does not stand in their way.

How would you customize a big blue button?

As explained when talking about functionalities, the configurable client features of BigBlueButton are split into different modules that can be loaded at startup [2]. There is a `config.xml` file that includes each loaded module along with all their configurations and all their dependencies [3].

```
<module name="WhiteboardModule" url="http://HOST/client/WhiteboardModule.swf?v=VERSION"

    uri="rtmp://HOST/bigbluebutton"
    dependsOn="PresentModule"
    baseTabIndex="601"
    whiteboardAccess="presenter"
    keepToolbarVisible="false"
/>
```

Above we can see an example of the `config.xml` structure where the WhiteBoardModule is declared. Here the module has a `dependsOn` field, which means that the WhiteBoardModule cannot be loaded unless PresentModule has already been loaded. An example of a configurable item is the `keepToolbarVisible` which can be set to `true` or `false` .

The usage of an xml file allows enough flexibility to the customer adopting the platform. In fact, this design choice does not ask the administrator of the system to edit the source code in order to enable or disable functionalities but only requires work on a human-readable XML file. These configuration settings are although intended to be used by the system administrator, and not by users like teachers or students. All these configurations are loaded at startup time which means that the program will not load new modules (or changes to the current ones) dynamically at runtime. However, at the same time, it will not require a full recompilation of the code to process the changes. This policy, from a customer perspective, can be considered as a good tradeoff in order to have a system that can change semi-dynamically. Apart from this strategy not many other expedients are used for aiding variability and evolution of the system. When it comes to using these configurations, methods use `if` checks to verify the settings specified in the XML file. This makes it more difficult to edit pre-existing modules but at the same time leaves space for future expansions with new features. In fact, building a new module requires just to add a new `<module>` section to the `config.xml` file. A final remark on customization is the branding allowed by BBB, where the skin of the system can be modified in a CSS file that is also compiled before running the client side, where logos or color schemes can be added [20].

How can it evolve?

BigBlueButton as an Open-Source Software is intrinsically subject to change. Furthermore, as a project dedicated to education, it has to be able to adapt to different and upcoming needs that may appear. The main team of developers at BBB are fully responsible for the assessment, planning and management of change but it is mostly triggered by outside factors.

Past and future

The BBB project has been in development for nearly 10 years and has evolved a great deal over that period. New features have been introduced in many of their 14 releases but also with some of the releases only focusing on stability and improvements [8]. The biggest milestones in evolution have come from explicit or implicit user needs, as well as strategic decisions by the team. Early additions to the software in 2009 includes packaging the software into a virtual machine in order to make it easy for users to set up BBB [12]. A big feature was then introduced in 2011 that enabled Record and Playback which had been widely requested by the community [13], which required extensive rework on the server architecture [5]. Later on, the team added accessibility support with screen readers for users with sight disabilities. The focus of the year 2015 was mostly dedicated to providing users with world class quality, such as having an excellent audio experience. The future goals for BBB include support for users with hearing disabilities (Closed Captioning), adding unit testing to core modules and creating a HTML5 client (which is currently under development) [9]. The HTML5 client will give users an option to join a session without having to use Flash, like it is required at this moment [10]. The reason why Flash has been used so far is because of the previous limitations of HTML5 regarding capturing webcam and audio [14].

Ease of evolution and tradeoff

Systems can be designed to be highly flexible and easy to change or simpler in the way that they meet the original requirements and might need extensive refactoring for later changes [11]. BigBlueButton is somewhere there in between but closer to being a flexible software.

The BBB team has taken into consideration certain aspects that make future evolution easier. The internal structure of the system and the various modules that are part of it facilitate the evolution and new features. Every implemented new feature is related to a module and as a result is integrated with a reliable existing base. When we interviewed Fred Dixon on this subject he said:

> The client was always modular and we kept that modularity through the development of the client (this is making it easy for us to add Closed Captioning, a feature planned for a future release).

This modularity has therefore been a part of the structure from the beginning. As a part of the internal structure of the modules, they include services that communicate with each other through broadcasted messages. As a result when new features are added, they can take advantage of all the events that are already broadcasted from the messages and build on them. Another way that eases evolvability is implementing parts of the system using third party softwares. It is very cost and time efficient and allows the team to focus on the core functionalities, but also leads to dependencies on these external softwares. Furthermore, with an iterative approach with many releases the BBB team allows its users to take advantage of new features as soon as they are ready and stable.

However, there are a few aspects of the system that can be considered as obstacles for evolution. Currently development is mainly supported on Ubuntu machines. The VM integration on non-Ubuntu machines can be a problem for developers and slow down the evolution of the system. Moreover, the BigBlueButton client is built on Flash using ActionScript. ActionScript is becoming outdated and there are some problems with the development of the client side related to the usage of Flash such as a difficulty of logging.

In general, the BBB team uses modularity to ease future evolution but also tackle some problems when they appear, all in all with a main goal of keeping the system completely stable.

But, hey! I want to contribute as well

The BigBlueButton community is very inviting for new developers to contribute to their project [14]. In order to uphold the quality of their development they require certain things from contributors and for them to apply a certain workflow [4]. They require contributors to first study the architecture of the system and fully understand it before adding or modifying any code. Before making a contribution it is also advised to have participated in the dev mailing list on Google Groups by showing understanding of the system and to build trust with the community.

When choosing what kind of contribution to make it is advised to tackle an open issue or if the contributor is not a developer, to submit translations for localization or, to help others with the installation and configuration of a BBB server.

If contributing with code, it is required to first sign a CLA and to announce your intent on the issue itself on GitHub or on the dev forum and receive positive feedback. When submitting the contribution it is necessary to follow a certain workflow on GitHub by forking the repository, creating a topic branch and submitting a pull request from that branch. If the patch/feature is big it is also necessary to a accompany the PR with test cases and to follow certain coding conventions.

Aaand...we are done

In this chapter we have analyzed the architecture of BigBlueButton based on different views and perspectives, so readers can have a broad understanding of the software and its structure.

In conclusion, BigBlueButton is an open source on-line conference system for education, where its developers strive to provide excellent tools and features for high quality remote learning. There is a strong community around the project where the core developers make an effort to reply quickly to the active mailing lists. Even though it is an open source project the managers of BBB treat it as a product, having specific development and release processes and focusing on good code quality. New contributors are highly welcomed and the core developers go out of their way to make the environment good by having excellent documentation available and good support on the mailing lists. The BBB team focuses on having the software very stable in each release which has attracted some heavyweight customers and third party integrations such as WordPress, Moodle and The Defense Information System Agency (DISA).

The structure of the code is very modular which eases evolution and helps to minimize technical debt. Many external components are used which allows the BBB developers to focus more on the client side of the project. As a part of keeping the project stable, it is heavily tested by the community before each release but they lack automated tests.

Analyzing this project has proven beneficial for us to learn about how a good structured software is. We got the opportunity to take part in a great open source community by making small contributions to the project where we were welcomed with inviting and helpful communication with the project manager Fred Dixon. This was a very valuable experience where we learned how to approach an open source project and it provided a good way to learn about the system's architecture. For other developers or people interested in open source projects, it is very interesting to study such a well established and stable project, and we hope this analysis can help with that.

References

1. 18f, https://18f.gsa.gov/2015/10/05/managing-technical-debt
2. Client Config, http://docs.bigbluebutton.org/dev/client-configuration.html#Client_Configuration_
3. config.xml, https://github.com/bigbluebutton/bigbluebutton/blob/master/bigbluebutton-client/resources/config.xml.template
4. BigBlueButton Documentation FAQ, http://docs.bigbluebutton.org/support/faq.html
5. BigBlueButton History, http://bigbluebutton.org/history
6. BigBlueButton downloads in July 2012, http://bigbluebutton.org/2012/07/06/bigbluebutton-is-a-global-project/
7. BigBlueButton Open Source Integrations, http://bigbluebutton.org/open-source-integrations
8. BigBlueButton Release Notes, http://docs.bigbluebutton.org/support/release-notes.html

9. BigBlueButton Road Map, http://docs.bigbluebutton.org/support/road-map.html
10. http://docs.bigbluebutton.org/labs/html5-overview.html
11. Rozanski and Woods. *Software Systems Architecture: Working With Stakeholders Using Viewpoints And Perspectives.* Upper Saddle River NJ: Addison-Wesley, 2012.
12. BigBlueButton talk at Carleton University, https://www.youtube.com/watch?v=FXNeNZqVwsQ
13. Response to a request for Recording on the user forum, https://groups.google.com/forum/#!searchin/bigbluebutton-users/record$20and$20playback%7Csort:date/bigbluebutton-users/c6fCmD4Lc3M/gcQrmrI1-dsJ
14. Floss Weekly, Interview with Fred Dixon, https://twit.tv/shows/floss-weekly/episodes/147
15. Bright Hub Project Management, *What Is the Power/Interest Grid?*, http://www.brighthubpm.com/resource-management/80523-what-is-the-powerinterest-grid
16. Technology Innovation Management Review, *Lessons from an Open Source Business* - Fred Dixon, http://timreview.ca/node/441
17. BigBlueButton Open Source Components, http://bigbluebutton.org/components
18. BigBlueButton Architecture, http://docs.bigbluebutton.org/overview/architecture.html
19. SQALE http://www.sqale.org
20. BigBlueButton Branding, http://docs.bigbluebutton.org/dev/branding.html
21. BigBlueButton GitHub issue 2517, https://github.com/bigbluebutton/bigbluebutton/issues/2517
22. BigBlueButton GitHub issue 973, https://github.com/bigbluebutton/bigbluebutton/issues/973
23. Moderator/Presenter Tutorial (0.9.0), https://www.youtube.com/watch?v=J9mbw00P9W0
24. Viewer Overview (0.9.0), https://www.youtube.com/watch?v=LS2lttmPi6A
25. Interview with Fred Dixon via email
26. BigBlueButton Contributors, https://github.com/bigbluebutton/bigbluebutton/graphs/contributors
27. BigBlueButton Blog http://bigbluebutton.org/blog/
28. Boston College STE Tweet https://twitter.com/BC_CTE/status/689859706370686980
29. C4ISR&Networks, *DISA to replace DCO with new collaboration services tool*, http://www.c4isrnet.com/story/military-tech/disa/2015/02/11/disa-to-save-12m-defense-collaboration-services/23238997/
30. BigBlueButton GitHub Commits https://github.com/bigbluebutton/bigbluebutton/graphs/commit-activity
31. BigBlueButton Overview http://bigbluebutton.org/overview/

Bootstrap

By Patrick van Hesteren and Floris Verburg
Delft University of Technology

Bootstrap is an easy-to-use and powerful front-end framework for faster and easier web development and it is the most popular HTML, CSS, and JS framework for developing responsive, mobile first websites. In this chapter, the Bootstrap project is analyzed by means of architectural views and perspectives. The context view, development view and functional view are provided, just as the evolution perspective and usability perspective. Lastly, possible improvements to the Bootstrap project are mentioned.

Introduction

Bootstrap is a sleek, intuitive, and powerful front-end framework for faster and easier web development. Bootstrap was created by Mark Otto and Jacob Thornton at Twitter mid-2010. Before it was a successful open-sourced framework, Bootstrap was known as Twitter Blueprint. During the first Hack Week of Twitter, after Bootstrap was a few months into development, the project exploded because developers of all skill levels jumped in without any external guidance.

The first official release of Bootstrap was on the 19th of August, 2011. Bootstrap has released over 20 releases since then, including two major rewrites with version 2 and version 3. The fourth version of Bootstrap had its first alpha release on the 19th of August, 2015 while the development of Bootstrap v4 is still in progress. Nowadays, Bootstrap is the most popular HTML, CSS, and JS framework for developing responsive, mobile first projects on the web [7].

Bootstrap is maintained by the founding team, Mark Otto and Jacob Thornton, together with the core development team and with the support and involvement of the Bootstrap community. The Bootstrap community consists of various people contributing to the project. The founding team and core team decide the direction Bootstrap is going. The core team and the community implement the functionalities together.

In this chapter, the insights of the Bootstrap project are given. We will give some background on the different stakeholders, architectural views, architectural perspectives and possible improvements to the project.

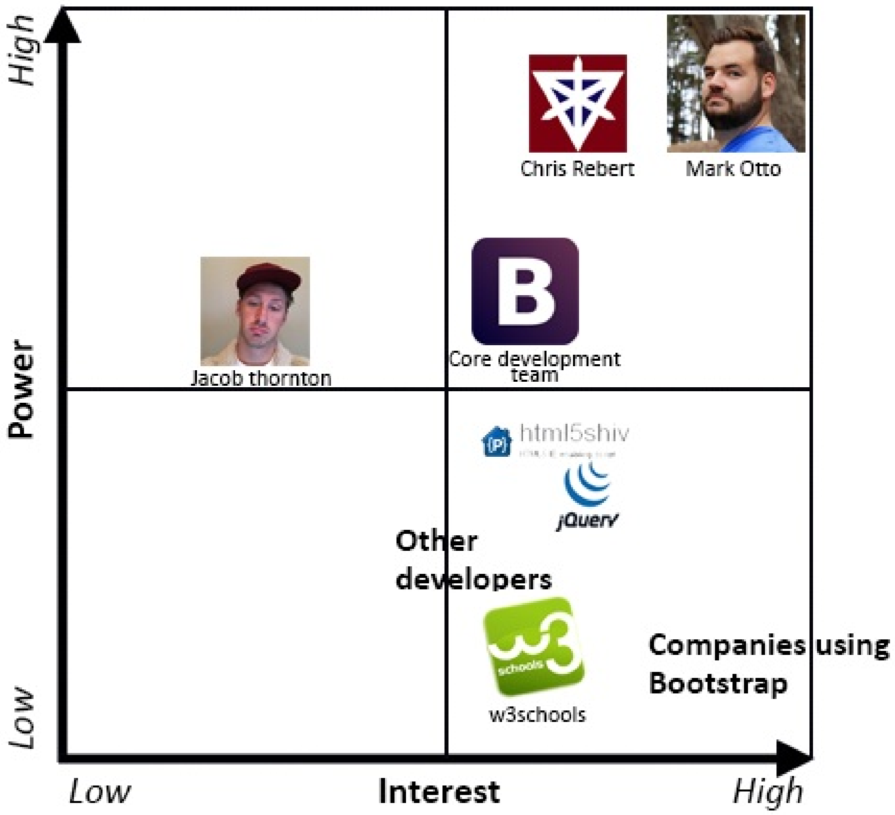
Stakeholders

Stakeholders are the people, groups and/or organizations that have interest or concern in an organization. Stakeholders can affect or be affected by the organization's actions, objectives and policies. Rozanski and Woods[2] discuss 11 types of stakeholders in their book. The most important ones for Bootstrap are listed below:

- **Acquirers** oversee the procurement of the system or product. The people that are involved in this stakeholder class are Mark Otto and Jacob Thornton.
- **Assessors** oversee the system's conformance to standards and legal regulation. Mark Otto is the only assessor in the Bootstrap project.
- **Communicators** explain the system to other stakeholders via its documentation and training materials. Persons involved in this class are W3Schools, Chris Rebert and random GitHub contributors.
- **Developers** construct and deploy the system from specifications, or lead the teams that do this. The core team develop the sytem and are the only people that can merge pull requests. Furthermore, random GitHub contributors are involved in this class.
- **Maintainers** manage the evolution of the system once it is operational. The core team are involved in this, mostly Chris Rebert.
- **Testers** test the system to ensure that is is suitable for use. The core team and random GitHub contributors write the test suite for Continuous Integration. All tests are automatically run upon committing to the repository.
- **Users** define the systems' functionality and ultimately make use of it. Companies and web developers use Bootstrap as framework for their website. A small selection of large companies that use Bootstrap is: Twitter, Walmart, NBA, Codecademy.

Power/Interest Grid

A widely used method to classify stakeholders is Mendelow's power/interest grid[1]. On this grid, the interest of the stakeholder is shown on the horizontal axis and the power of the stakeholder is shown on the vertical axis. The higher the power of the stakeholder, the more important it is the stakeholder is satisfied in the decision making. The higher the interest of the stakeholder, the more the stakeholder would like to be kept informed. The stakeholders with high power and interest must be managed closely and the stakeholders with low power and interest must be monitored but with minimum effort. The power/interest grid of the Bootstrap project is shown below.



Bootstrap's Power Interest Grid

The person with the highest interest and power is Mark Otto, one of the founders of Bootstrap and still one of the most active developers of the project. Someone close to him is Chris Rebert, he is not one of the founders but he is the person from the core development team that is most active and he reviews and handles most of the pull requests.

Jacob Thornton is also one of the founders, so he still has quite a lot power but he is currently not so active anymore. Therefore, he has less interest in the project.

The core development team is also a stakeholder with high interest and power. These are the people that actively contribute to and maintain the project and have write permissions to the project.

The companies that use Bootstrap - especially the big ones - are important to keep satisfied. If these companies are no longer satisfied, they will stop using Bootstrap. If no one uses Bootstrap anymore, the project will lose its value. Therefore, it is important for Bootstrap to keep these stakeholders satisfied.

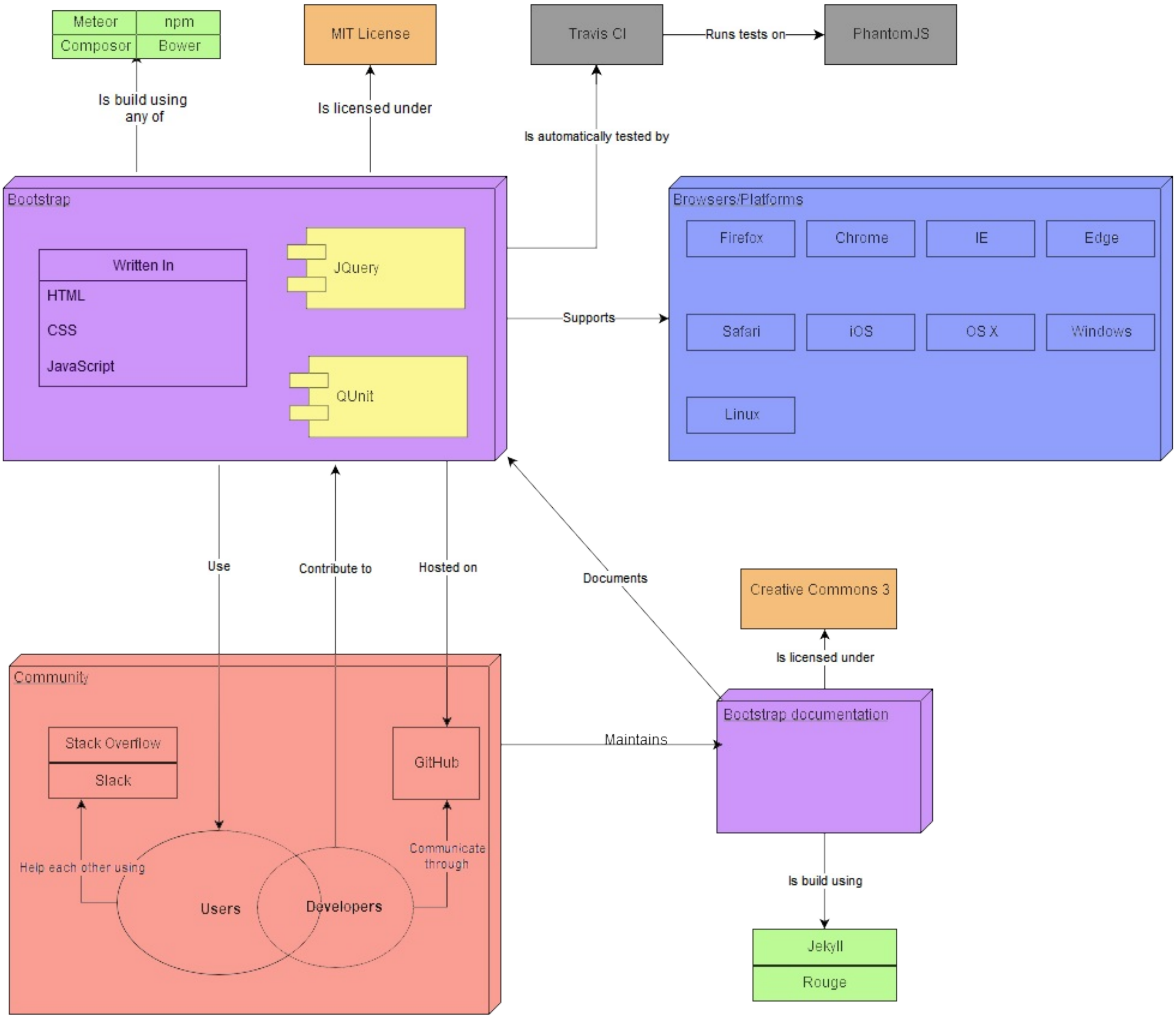
Stakeholders like W3Schools and jQuery have high interest in the project. It is important for Bootstrap to keep these stakeholders informed. W3Schools is a source of information about Bootstrap that a lot of users use. jQuery and also for example html5shiv, are also important to keep informed. Because Bootstrap depends on these stakeholders, the stakeholders must know if Bootstrap wants to change anything that depends on for example jQuery.

Context view

The context view of a system defines the relationships, dependencies and interactions between the system and its environment. This environment includes the people, systems and external entities with which it interacts. It defines what the system does and what the system does not do [2].

Context Model

Bootstrap's context model is visualized in the following figure and will be explained in depth in the following section.



Bootstrap's context model

Next to the code written by the contributors of the project (purple), Bootstrap is built using the jQuery library (yellow) and the coupling with this library is very strong. If a backwards incompatible change is made, it requires a fair bit of work to support the new version. Bootstrap does its dependency management through Grunt, which depends on Node.js. Grunt builds Bootstrap, the tests and the documentation. The framework itself is built using Meteor, npm, Composor or Bower and the documentation is built using Jekyll and Rouge (green).

On the testing side, the Bootstrap's unit tests are written in QUnit (yellow). This dependency is fairly strong as migrating away from QUnit (and having maintainable tests) would be a fair bit of work. The automated testing infrastructure (grey) consists of Travis CI running the tests on PhantomJS. It also runs JSHint for code analysis. This dependency is not as strong as it is in the infrastructure instead of the code.

One of the functions of Bootstrap is to abstract away the differences between different platforms and browsers (blue). This makes all of the supported combinations an external dependency. When a backwards incompatible change is made, Bootstrap has to adjust to support the change. Bootstrap is a framework that supports all large browsers and mobile platforms.

For the users of Bootstrap looking for support, StackOverflow and Slack are used, which are part of the community of Bootstrap (red). These are only loosely coupled with the project and different platforms could be substituted in easily.

Bootstrap's source code is also licensed (orange) under the MIT license. Its documentation is licensed under creative commons 3 with the notable exception of the code fragments, which are licensed under the MIT license.

Development view

The development view provides an insight in the way Bootstrap is being developed. The development view is the view that addresses the specific concerns of the software developers and testers. First we will explain the architecture of the project and thereafter, we will discuss the technical debt.

Dependencies

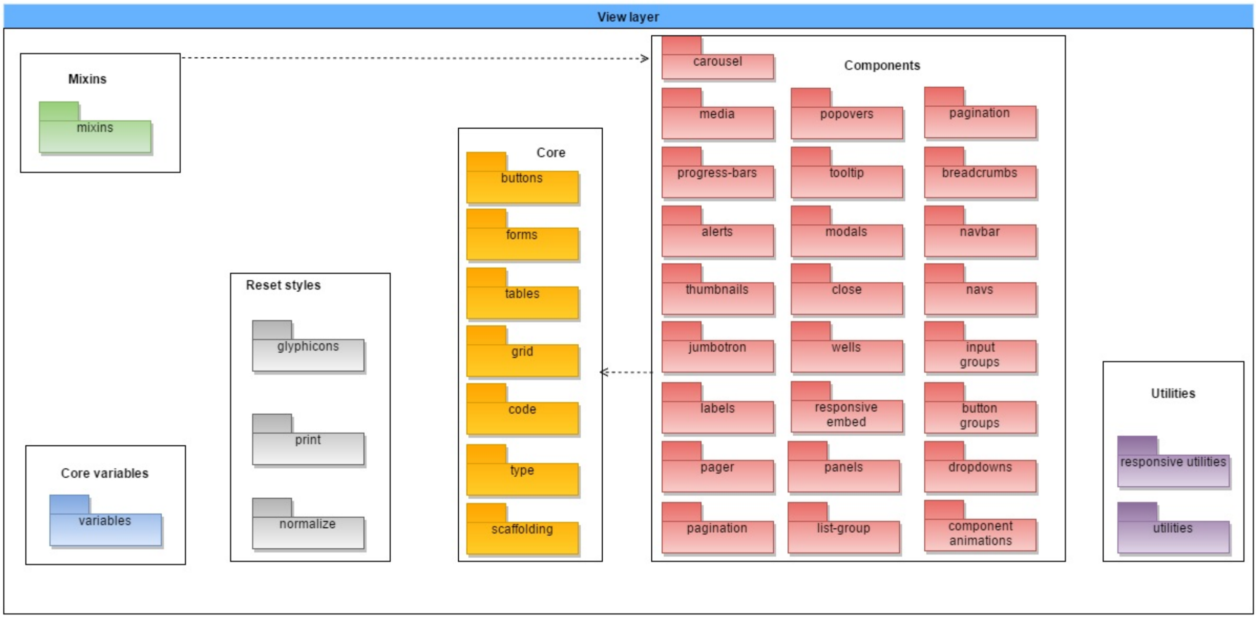
In order to reduce the amount of dependencies, Bootstrap aims to be modular by introducing new functionalities as plugins. As described in the development view section, Bootstrap hardly has any dependencies between the models, which indicates low coupling. In the case of Bootstrap, this does not lead to a high cohesion, as is usually the case with large projects, since all components are stand-alone.

Source code modularization

Bootstrap is designed in a modular way in order to reduce the total amount of dependencies within the system. Bootstrap is built using two different components: the logic layer and the view layer, which are explained below. We can summarize Bootstrap's architecture as a view-view-controller architecture. On the one hand, we have the views, which are used for visual display. On the other hand, we have the view-controller that describes the behavior of all visual components (e.g. disabling a button after it has been clicked). When creating a webpage with Bootstrap, designing it in the popular *Model-View-Controller*[3] way would make the most sense. When using Bootstrap within a web application using the Model-View-Controller design principle, Bootstrap will serve as the view component.

View layer

The view layer consists of 6 different modules: the core variables (blue), reset styles (gray), core (yellow), components (red), utilities (purple) and finally the mixins (green). The architecture of the view layer can be seen below.



The architecture of the view layer

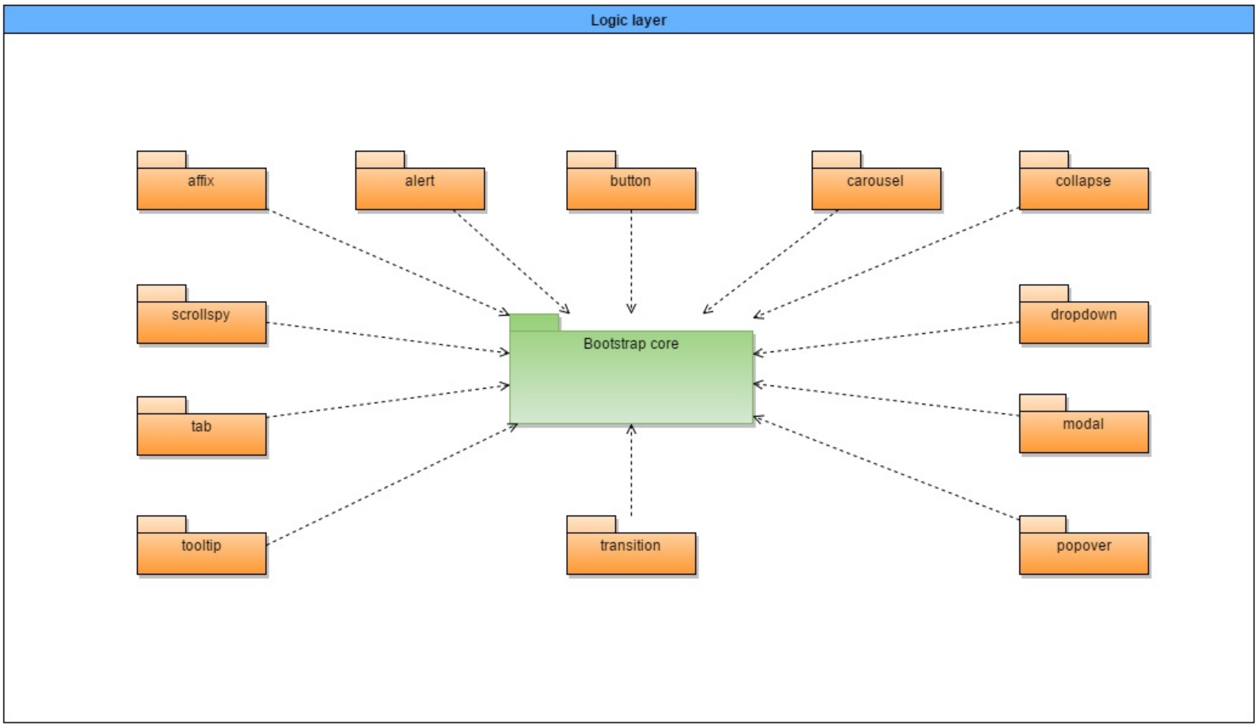
- The core variable module consists of a style file that loads all different global styles that are used within Bootstrap.
- The reset styles module has the purpose of resetting or overriding the default browser styles to ensure that a Bootstrap page looks similar across all different device/browser combination styles that may occur when accessing a webpage built using Bootstrap.
- The core module consists of the backbone of Bootstrap and contains functionality that is

generally used within all webpages built using Bootstrap.

- The components module consists of commonly used components within webpages that are not available in HTML by default. All components have been built independently and are only dependent on the core module.
- The utilities module is responsible for one of the most popular features of Bootstrap: it's responsiveness and Mobile First way of designing.
- The mixins module does provide a way to quickly generate simple, semantic layouts. Mixins are built on top of the core and components and reuse these to generate the semantic layouts which can be adjusted by overriding their properties.

Logic layer

The logic layer provides the logic that all visual aids, the Bootstrap components, require. The architecture of the logic layer can be seen below.



The architecture of the logic layer

The logic layer consists of 12 different components that each provide a different functionality for a different corresponding visual aid. All different components rely on the Bootstrap core, which verifies that the jQuery dependency is loaded and provides CSS transition support.

As shown in the diagrams above, all modules have been built as independent as possible in order to reduce technical debt. While downloading, Bootstrap includes all modules by default. It is also possible to manually remove these from the includes in order to reduce resource usage within the web application.

Technical debt in Bootstrap

The absence of a clear definition and model for technical debt can exacerbate the challenge of its identification and adequate management. It can further prevent the realization of technical debt’s utility as a conceptual and technical communication device[5].

Technical debt can be divided into multiple dimensions: code debt, design and architectural debt, environmental debt, knowledge distribution and documentation debt and testing debt[5]. We have measured Bootstrap's technical debt using these dimensions. Measuring the technical debt from Bootstrap differs from other projects as it is built on the idea of providing independent plugins. Furthermore, CSS files cannot be measured in terms of complexity.

Code debt

Code debt has to do with unnecessary code duplication and complexity, bad style that reduces the readability of code and poorly organized logic that makes it easy for a software solution to break when updated at a future point in time. We have ran the JavaScript source files against complexity-report. This produces the following metrics:

- Mean per-function logical LOC (lower is better): 5.865373712711182
- Mean per-function parameter count (lower is better): 0.6435881047084829
- Mean per-function cyclomatic complexity (number of cycles in the program, lower is better): 1.867627071212864
- Mean per-function Halstead effort (effort required to write the program): 2373.7342178742506
- Mean per-module maintainability index (-infinity - 171, higher is better): 117.82584093099538
- First-order density (percentage of all possible internal dependencies that are actually realized in the project, lower is better): 0%
- Change cost (percentage of modules affected, on average, when one module in the project is changed, lower is better): 3.8461538461538463%
- Core size: (percentage of modules that are both widely depended on and themselves depend on other modules, lower is better): 0%

All metrics are defined on a linear scale, apart from the maintainability index which is defined on a logarithmic scale. While there is no such thing as a 'normal' score for these metrics, since every project is composed very different, it is however immediately show that Bootstrap has a very low amount of dependencies and changing a module would hardly require any changes to be made to other modules. Furthermore, we see that there is no unnecessary complexity introduced (low amount of parameters in functions, low amount of lines of code per function, low amount of cyclomatic complexity), which eases the maintainability of Bootstrap. It is clear that there is hardly any code debt in the Bootstrap project.

Design and architectural debt

Design debt has to do with upfront design with a lack of focus on for example maintainability and adaptability. Architectural debt has to do with sub-optimal solutions, such as the usage of superseded technologies or patterns.

Bootstrap works with an extensive issue and shipping list. While the issues are created ad-hoc, the shipping list is defined months in advance. New features are thought of in advance and all changes that are made to the system are verified to be in line of the upcoming releases (e.g. issue #19303). New features can be implemented at once, but will however not be added to pre-defined shipping lists. Instead, they will be added into feature releases. Bootstrap has chosen to work with technologies that are stable and supported. If better solutions become available, a switch is made in the next release.

Overall, the design and architectural debt of Bootstrap is very minimal, as we have not stumbled upon any outstanding issues that would have impact on the design/architecture of the upcoming or current releases.

Environmental debt

Environmental debt of an application has to do with the environment in which the application is hosted or developed, especially the manual tasks and the frequency in which they occur. Since Bootstrap is a framework and is not deployed on its own, there are hardly any manual or incidental tasks that need to be performed within the environment in order to keep developing the framework. Since most of the issues are created by random contributors and most bug-fixes and feature implementations are made by them as well, the Bootstrap core team has chosen to take up an overseeing role. Since a lot of issues and pull requests are created every day, all must be verified according to code standards and checked whether or not the changes made are in line with feature releases.

We see that the Bootstrap core team has to invest a lot of time in maintaining and informing the community about the changes. The environmental debt of Bootstrap is as minimal as it can be and is mostly caused by the popularity of Bootstrap, since this triggers a lot of community involvement that needs to be managed properly. This cannot be reduced unless the core team will start to implement more features themselves.

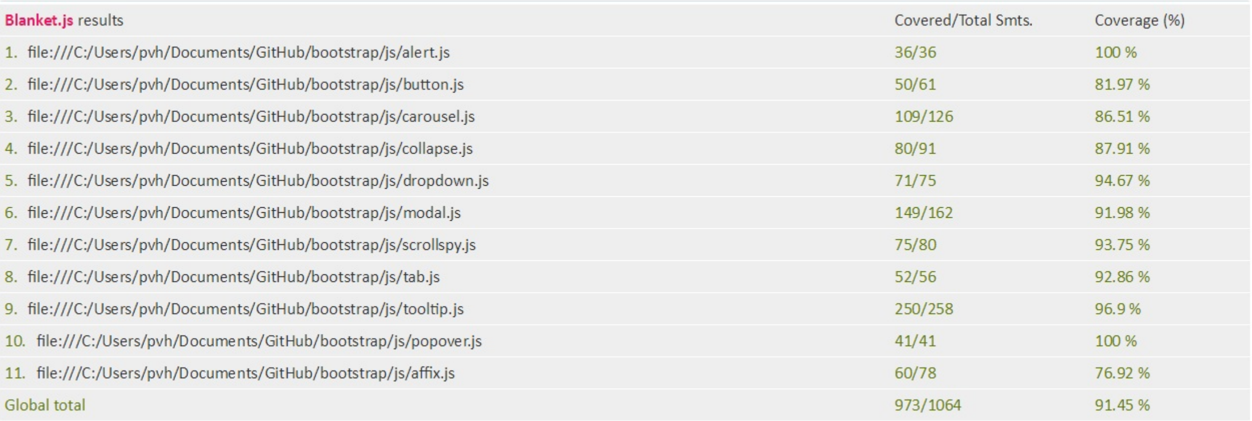
Knowledge distribution and documentation debt

Knowledge distribution and documentation debt has to do with the way the knowledge and documentation of the project is spread / maintained over time. Bootstrap has an extensive documentation and in general, each of the different components of the framework is described with all different options plus working examples. The documentation is available via GitHub and via the official Bootstrap website, making it very accessible for knowledge sharing. The community actively contributes to the documentation, especially to documentation for new features that have not been documented properly yet. Since Bootstrap has a lot of random contributors, it has an extensive documentation on how to build and contribute to the project. The knowledge distribution and documentation debt of Bootstrap is therefore very minimal.

Testing debt

Testing debt measures the extent to which an application is tested properly in order to ensure that the application keeps functioning when changes are made.

We ran BlanketJS on the QUnit test suite of Bootstrap in order to measure the code coverage of the JavaScript files. Below, you can find the results.



Blanket.js results	Covered/Total Smts.	Coverage (%)
1. file:///C:/Users/pvh/Documents/GitHub/bootstrap/js/alert.js	36/36	100 %
2. file:///C:/Users/pvh/Documents/GitHub/bootstrap/js/button.js	50/61	81.97 %
3. file:///C:/Users/pvh/Documents/GitHub/bootstrap/js/carousel.js	109/126	86.51 %
4. file:///C:/Users/pvh/Documents/GitHub/bootstrap/js/collapse.js	80/91	87.91 %
5. file:///C:/Users/pvh/Documents/GitHub/bootstrap/js/dropdown.js	71/75	94.67 %
6. file:///C:/Users/pvh/Documents/GitHub/bootstrap/js/modal.js	149/162	91.98 %
7. file:///C:/Users/pvh/Documents/GitHub/bootstrap/js/scrollspy.js	75/80	93.75 %
8. file:///C:/Users/pvh/Documents/GitHub/bootstrap/js/tab.js	52/56	92.86 %
9. file:///C:/Users/pvh/Documents/GitHub/bootstrap/js/tooltip.js	250/258	96.9 %
10. file:///C:/Users/pvh/Documents/GitHub/bootstrap/js/popover.js	41/41	100 %
11. file:///C:/Users/pvh/Documents/GitHub/bootstrap/js/affix.js	60/78	76.92 %
Global total	973/1064	91.45 %

JavaScript code coverage of Bootstrap

We notice that while there is some room for improvement, all JavaScript files are covered quite well and there is no real testing debt here.

In addition, there are some visual tests containing predefined webpages that should look correct when testing the project. These need to be ran manually though.

The test pyramid [6] as described by Martin Fowler is however not properly adhered to. While we admit that the majority of the framework consists of CSS files which are hard to test, one could however easily automate the testing of the proper scaling of the grid system when resizing a webpage using Nightwatch for example. Also, the integration of different components should be tested better, as Bootstrap is only tested at unit level and very basically at a visual level.

Prevention of technical debt

Contributors need to be are aware of the fact that they are not the only one using and/or contributing to the system and that there is a common baseline to which all contributions must adhere in order to reduce technical debt. Only by introducing technical debt measurements, a project like Bootstrap will remain maintainable. Next to the measurements regarding the source code, which are explained below, the documentation also has its own guidelines.

Code guidelines

The Bootstrap project uses HTML, CSS and JavaScript, which each have different code guidelines. Mark Otto, one of the creators of Bootstrap, has provided standards for developing flexible, durable, and sustainable HTML and CSS where contributors of the project must adhere to. This code guide can be found here. In this code guide, the preferred syntax of HTML and CSS are defined, just as some rules the code must abide by. Next to this code guide, Bootstrap provides its own guidelines for every language.

Test guidelines

If a contribution to the project contains JavaScript code, a set of relevant unit tests must also be provided. Each Bootstrap plugin of the project has its own dedicated test suite.

Next to these unit tests for each Bootstrap plugin, the project also has third-party testing-related code, which are written for QUnit and jQuery. And lastly, the project has "visual" tests, which are run interactively in real browsers and require manual verification by humans.

Automated Testing Infrastructure

In order to be able to ensure the quality and integrity of the Bootstrap project throughout the entire development phase, Bootstrap also uses automated testing through continuous integration. The different parts of the continuous integration are Travis CI and Hound CI.

Travis CI

Travis CI is a continuous integration service that is used to build and test software projects that are hosted on GitHub. Every time a pull request is opened or something is pushed to a branch in the Bootstrap project, Travis CI automatically runs all the unit tests of the project. First, the framework and documentation are both generated. Then each of them is validated/tested.

Hound CI

In addition to Travis CI, Hound CI is also automatically triggered when a pull request is opened. Hound CI is a hosted service that comments on style violations in GitHub pull requests. This allows the project contributors to better review and maintain a clean codebase. This is done in order to emphasize care and team communication and it encourages other developers working in the code to keep the campground clean. It also avoids unnecessary debate during code reviews.

Variability perspective

The extent to which a project can be configured of varied is called the variability of a project. In this section, an analysis of the variability of the Bootstrap project is provided.

Since Bootstrap is a framework, it cannot be configured like other software projects. We will therefore describe how each of the features makes use of other features and how they can be configured throughout the framework and describe the variability perspective from the users' point of view.

Variability / configuration strategy

As we have mentioned before, all Bootstrap functionality has been built on the idea of providing plugins that can be enabled/disabled based on the user's preferences. While by default Bootstrap will contain all functionality, it is possible to customize Bootstrap to your needs in order to make your website use less resources and save user's bandwidth in the end. While it is possible to compile the project on its own, it is also possible to customize and download your Bootstrap configuration via the customize page. This page is made available for less experienced users and eases the configuration of the framework.

Configuring the features

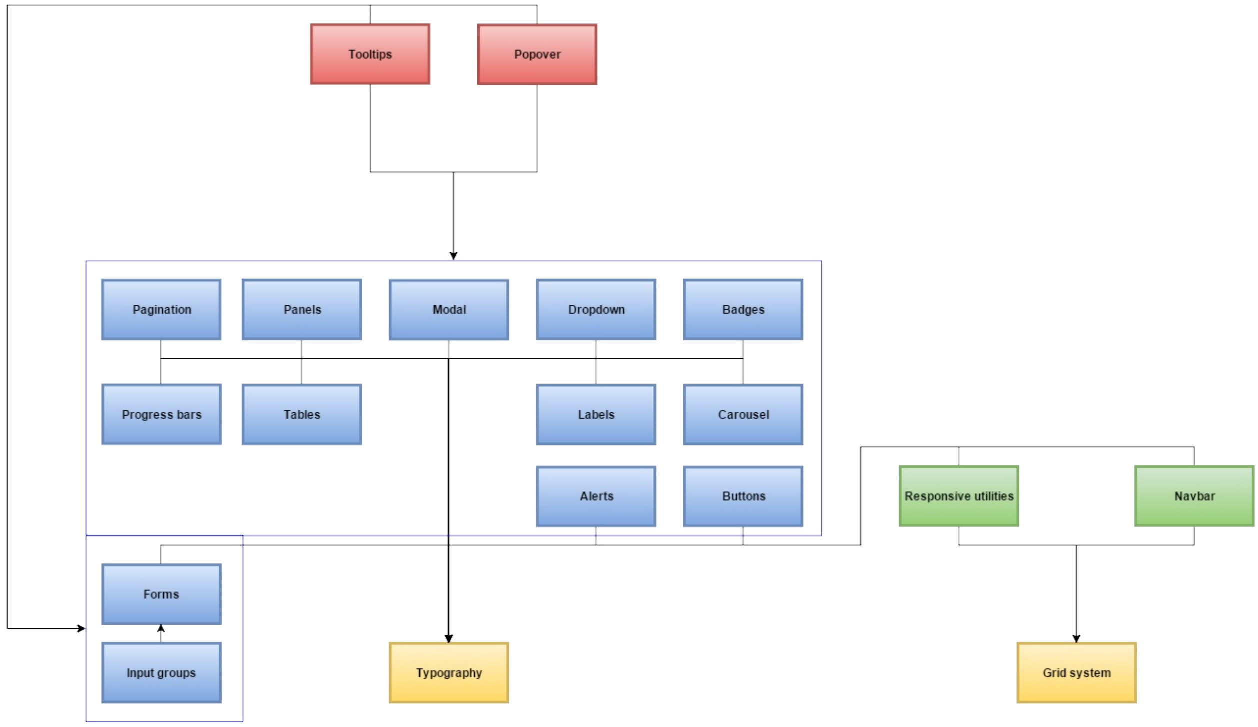
The configuration of Bootstrap differs from most project in the sense that it is a framework and not so much a stand-alone project. Therefore, the configuration opportunities lie within using (or not using) certain components in a certain way that suits your goals.

The customize page allows users to determine the features available throughout the project and to customize them. This page splits them up into two types: the style files and the jQuery plugins. One should of course make sure to exclude both the style sheet and the jQuery plugin when excluding a feature.

One is also able to exclude plugins or style sheets by recompiling the project after altering the style sheets or removing the mappings of the source folder to the distribution folder for both the style sheets and the plugins.

Relationships and dependencies between the different features

Bootstrap is very loosely coupled and just as with the source code modularization there are not many features that are dependent on one another. While in theory Bootstrap has been built on the idea of providing plugins that can be enabled or disabled, one cannot simply use a single plugin and neglect the rest of the plugins while still maintaining the *intended* functionality of the Bootstrap developers. The following diagram visualizes the dependencies among the different features, which are explained below. Please note that this diagram has been drawn as seen from the *intended* use of the Bootstrap developers.



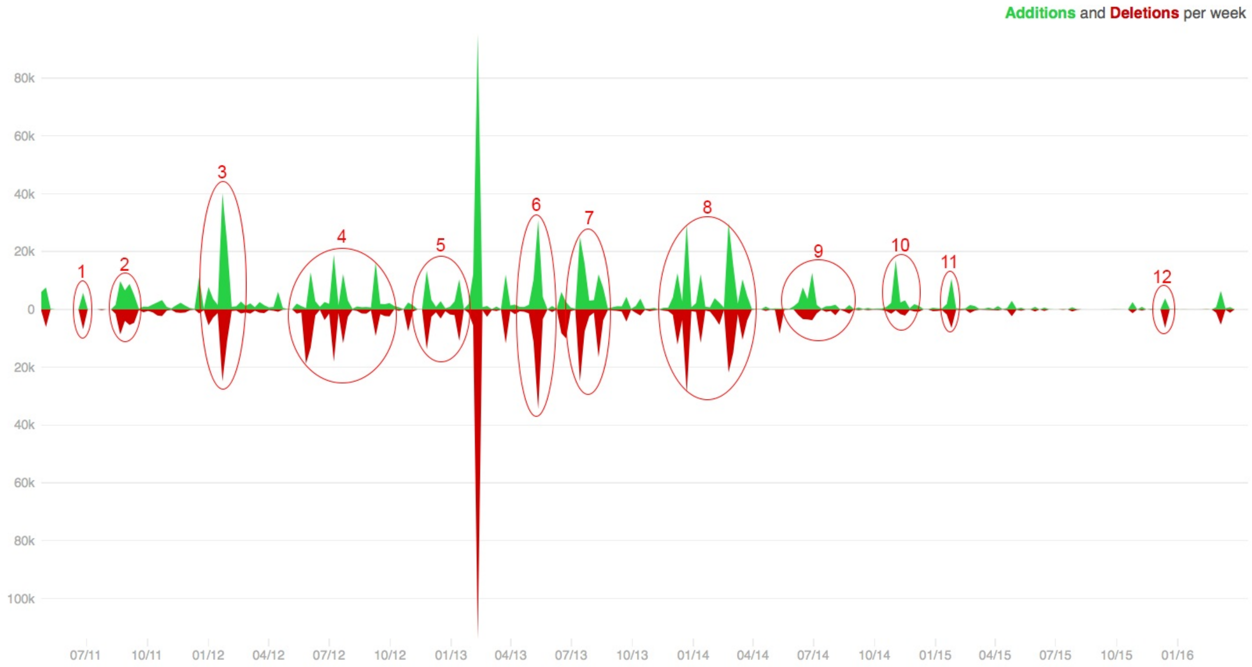
Dependencies among different features

- The key features (yellow) are the features that most other features are dependent on. These include the basic Bootstrap styles (typography) and the grid system.
- The navigation bar and the responsive utilities (green) are dependent on both key features as they both make use of the grid system and basic styles.
- Most components are modular (blue) in the sense that they have almost no dependencies, apart from the basic Bootstrap style. In addition, we see that the input groups are dependent on the forms feature as the input groups are to be used within the forms.
- Finally, the tooltip and popover (red) are dependent on almost every other modular

component (blue). While they do not depend on all components at once, the tooltip and popover features can only be used in combination with any of the modular components.

Evolution perspective

Throughout the years, Bootstrap has evolved a lot. At this moment, the project has 36 releases, from the release of v1.0.0 on August 18 2011 to the v4.0.0-alpha.2 release on December 8 2015. Between all of those releases, a lot of features has been added, as is also reflected by the increased size of the project. In the next figure, the additions and deletions per week are shown.



Additions and deletions per week

As can be seen in this figure, there are a lot of peaks in the additions and deletions. These peaks correspond to the releases of the Bootstrap project. The large peak, around February/March 2013 can't be explained through one of the releases, but there was probably a change in project structure or some mistake was made. The other peaks are explained one by one below:

1. The first release of Bootstrap, v1.0.0. This was the most basic version with only the key CSS features and minor documentation.
2. The release of v1.3.0. In this version, the first JavaScript plugins were added and the documentation was improved a lot.
3. The release of v2.0.0. This was basically an entire rewrite of the library. Also, additional responsive CSS for nearly all components was added.
4. Releases of v2.0.4 until v2.2.2. In these releases, the documentation and HTML and CSS were improved and the support of Travis CI, the continuous integration, was

added.

5. The addition of some features, the shift from global to local dependencies and the addition of the changelog in the repository.
6. Release of v2.3.2. The large amount of added and deleted lines of code is mainly caused by the update of the jQuery dependency.
7. The release of version 3 of Bootstrap. In this version, Bootstrap introduced a new, flat design. Also, Bootstrap is from version 3 on, thanks to a massive redesign and rebuilt, mobile first and always responsive. Also, the grid system is improved a lot in order to be optimized for different screen sizes and the JavaScript plugins are rewritten.
8. Releases of v3.1.0 and v3.2.0. The most important changes in these releases were the change of build system and packaging and the deprecation of some classes. Furthermore, responsive embeds and responsive utility classes were added, just as an extra bot checking live code examples in issues and pull requests on GitHub.
9. The release of v3.3.0. In this release, another extra bot was added for checking new pull requests for common mistakes. Also, some Less variables were added for easier customization and the accessibility of some plugins was improved.
10. A release of Bootstrap with mostly bug fixes.
11. A release of Bootstrap with mostly bug fixes.
12. Release of v3.3.6. This is the current release of Bootstrap.

Since mid-2014, less additions and deletions are done compared to earlier. This is because in the first few years, the framework had to be developed from scratch. Nowadays, the framework is well developed and has a lot of features. However, bug fixes are still applied and new features are still added.

Bootstrap 4 is currently in development. But because this isn't committed to the master branch, this isn't visible on the graph. This is an example that the framework continuously is improved according to the new standards in programming. While doing this, Bootstrap ensures a high quality framework with up to date dependencies. All of this ensures that Bootstrap continues to adapt while keeping in mind accessibility, making it an excellent choice for a web-development framework, also in the years to come.

Usability perspective

Applying the usability perspective ensures that the system allows those who interact with it to do so effectively [2]. The only place users may interact with the system is when a user creates a website with the Bootstrap framework. This interaction only happens via computers, because users don't develop websites on their mobile phones. Furthermore, HTML, CSS and JavaScript can be executed on every operating system, so Bootstrap doesn't need to cope with the variation in bandwidth, hardware capabilities, and rendering software issues.

The interaction between the user and Bootstrap doesn't occur via a user interface, but via predefined HTML, CSS and JavaScript notations. Therefore, Bootstrap is a framework that supports all large browsers as well as mobile platforms in order to be available to as many users as possible.

The only touch point for the Bootstrap project is the development of websites, which has a comprehensive touch point interaction. The user builds his website with Bootstrap and mainly the use of the grid system requires some knowledge in order to be able to use. Furthermore, the different components can require quite some time to adapt to your needs as well. Bootstrap tackles possible implementation problems with its extensive documentation.

Bootstrap users vary from unexperienced web developers to very experienced web developers. This can cause problems, because the unexperienced users want to develop their website as easy as possible, while the more experienced users want to tweak the Bootstrap framework. The Bootstrap framework can be used without any tweaks, but can also be adapted by the users to fit their needs. Components of Bootstrap can be excluded from the resources and a minified version of Bootstrap is available as well in order to minimize the resource usage and bandwidth, as is also explained in the variability perspective.

Possible improvements

Based on our research on the Bootstrap project, we have have come up with some suggestions to further improve the project. In order to make the project even more accessible, we suggest that the following changes are made to the project. These suggestions have also been submitted to the issues list of the Bootstrap project.

Extending the annotations of the style sheets and their implications #19418

To allow for manual adjustment to the developers needs, we recommend that all style sheets and their implications are documented so that developers become aware of the changes they are making and the possible implications they may have.

Development of a real-time implications view #19419

Often developers will want to change some styling specifically for a certain use case on their website. The Bootstrap customize page allows for these changes to be made, but when making these changes the implications are not shown. Bootstrap should allow developers to upload their webpage and to make the changes on the styling in split screen. On the left side, changes can be made, whereas on the right side their webpage will update accordingly so that changes can be made more easily.

Conclusion

The goal of this research was to study the stakeholders, architecture, configurability and usability of the well-known Bootstrap framework. Mark Otto, Chris Rebert and the Bootstrap core team belong to the most influential stakeholders, but companies using Bootstrap, jQuery and W3Schools are also important stakeholders of the project.

All Bootstrap functionality has been built on the idea of providing plugins that can be enabled/disabled based on the user's preferences. It is very loosely coupled and there are not many features that are dependent on one another. Bootstrap can be styled and customized using the customize page or via editing the source files and recompiling the project. This ensures the framework is accessible for both experienced and unexperienced users.

There is hardly any technical debt in Bootstrap, although the integration of different components could be tested better. Bootstrap aims to reduce technical debt using its automated testing infrastructure which uses Travis CI, Grunt, Sauce Labs, Amazon C3 and Hound CI. Furthermore, it provides a set of code and documentation guidelines.

Bootstrap has evolved a lot throughout the years. The last time, less additions are done, because Bootstrap is a full grown framework nowadays. This doesn't mean the framework isn't maintained or is slowly dying. Still a lot of bugfixes are applied and features are added.

To conclude, Bootstrap is a robust framework that allows developers to create responsive web applications in a very easy way. With many more features to come in the upcoming release, Bootstrap will continue to ease web development for years to come, so it remains an excellent choice for a web-development framework.

References

1. Olander, Stefan, and Anne Landin. "Evaluation of stakeholder influence in the implementation of construction projects." International journal of project management 23.4 (2005): 321-328.
2. Nick Rozanski and Eoin Woods. 2011. Software Systems Architecture: Working with

Stakeholders Using Viewpoints and Perspectives. Addison-Wesley Professional.

3. Burbeck, Steve. "Applications programming in smalltalk-80 (tm): How to use model-view-controller (mvc)." Smalltalk-80 v2 5 (1992).
4. Sven Apel, Don Batory, Christian Kästner, Gunter Saake. Feature-Oriented Software Product Lines. Springer, 2013.
5. Tom, E., Aurum, A., & Vidgen, R. (2013). An exploration of technical debt. Journal of Systems and Software, 86(6), 1498-1516.
6. TestPyramid. (n.d.). Retrieved March 27, 2016, from http://martinfowler.com/bliki/TestPyramid.html.
7. Official Bootstrap website, http://getbootstrap.com.

CKAN: The open source data portal

By Andy Chiu, Boyang Tang, Jihong Ju, Bo Wang

Delft University of Technology

Abstract

Pirates value gold as much as researchers value data. They crawl through the entire web driven by their unsatisfying hunger for more data. Every once in a while, they find a CKAN instance which is a wonderful treasure trove filled with data, free for everyone to take. As part of the Delft Students on Software Architecture book, we provide an in-depth analysis of CKAN the open source data portal. This is done by identifying key stakeholders and putting them into context. Followed by the software architecture and how to contribute. Finally, we end with an evolution perspective and information viewpoint.

Table of Contents

Introduction

As a researcher you can never have enough data, especially when facing socio-economic problems. Technological advancement has created the opportunity to make data freely available to everyone. Collaborating by sharing data on a world-wide scale is one step towards a unified front for meeting the challenges of tomorrow. Open data is the formalized definition of this idea. In practice, most data is kept under control by both public and private organizations due to commercial interest. Advocates of open data argue that these restrictions are against the communal good and that these data should be made available without restriction or fee. Furthermore, they should be re-usable without requiring permission so that we can build upon earlier work.

The Open Knowledge Foundation introduced CKAN, the open data portal. Since its release it has been deployed by many governments, organizations and companies to make their data open and available. It is a powerful data management system that provides the tools for publishing, sharing, finding and using data. There exist other data hub software that can contain collections of data from multiple sources, but they do not provide the same degree of freedom and flexibility as CKAN. The latter is an open source initiative that lets you avoid long-term lock-in and the code is freely adaptable. Furthermore, it contains a rich set of features that helps gathering data from multiple sources, faceted search, machine interface to data and metadata, and sharing public data with other CKAN instances.

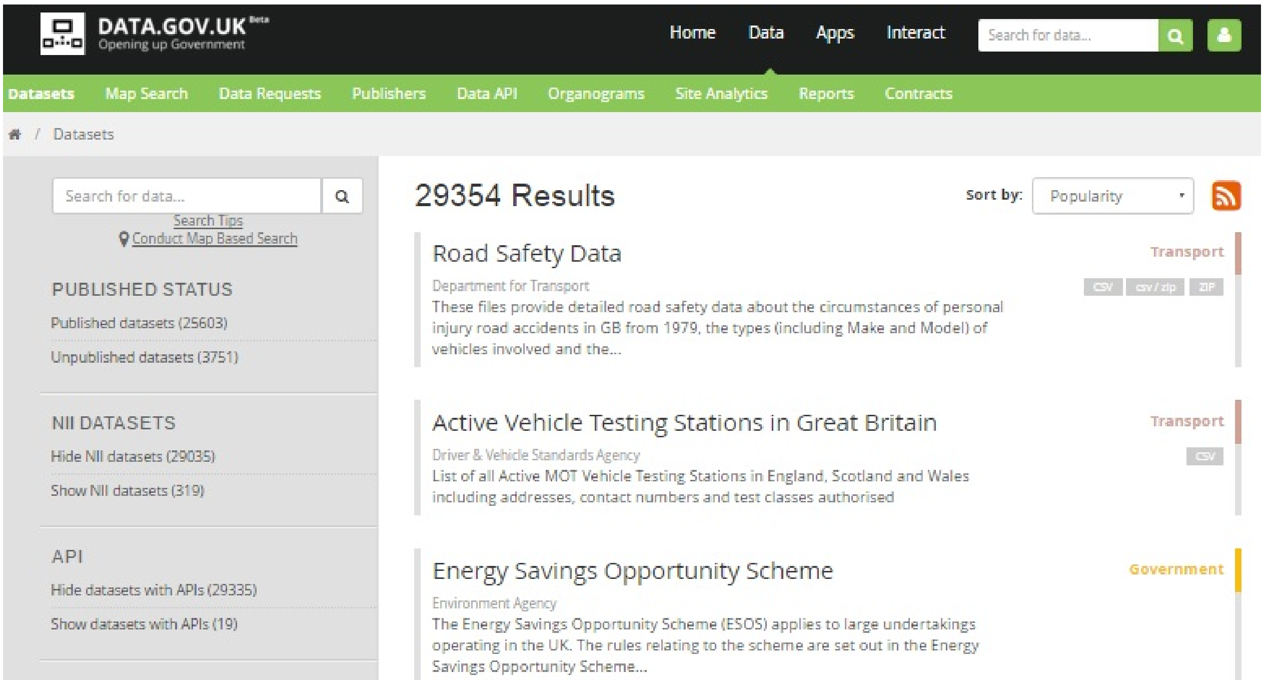
One of the many CKAN websites is data.gov.uk, they are also part of the Steering Group that drives the CKAN project forward. The purpose for launching data.gov.uk is to help people understand how the government works and how effective policies are made by releasing public data. Over 19000 data sets have been published so far by various UK departments. They encourage technical users to create useful applications out of raw data that can benefit society, or investigate how effective policy making changes over time.

In this chapter of Delft Students on Software Architecture we will provide an in-depth overview of the people involved in the CKAN project, the development process, how CKAN evolves over time and its information management.

Usability Perspective

A CKAN instance serves as a place where data can be published and searched. The success of the system depends on the effectiveness of these tasks. Therefore, applying the usability perspective can ensure that the system is well suited to user needs. This perspective will also give the reader an impression of how a CKAN instance looks like by introducing several key features and how these facilitate high usability. CKAN features can be either accessed by an intuitive web interface or by other machines through a common

API , here we mainly discuss the web interface. It is important to point out that there is a clear separation of the user interface and the functional processing. Figure 1 presents a screenshot from data.gov.uk, an exemplary data portal powered by CKAN. Visitors for such a website are mostly researchers or individuals with intermediate computer experience.



PUBLISHED STATUS

Published datasets (25603)

Unpublished datasets (3751)

NII DATASETS

Hide NII datasets (29035)

Show NII datasets (319)

API

Hide datasets with APIs (29335)

Show datasets with APIs (19)

29354 Results

Sort by: Popularity

Road Safety Data — Transport — CSV, csv / zip, ZIP

Department for Transport

These files provide detailed road safety data about the circumstances of personal injury road accidents in GB from 1979, the types (including Make and Model) of vehicles involved and the...

Active Vehicle Testing Stations in Great Britain — Transport — CSV

Driver & Vehicle Standards Agency

List of all Active MOT Vehicle Testing Stations in England, Scotland and Wales including addresses, contact numbers and test classes authorised

Energy Savings Opportunity Scheme — Government

Environment Agency

The Energy Savings Opportunity Scheme (ESOS) applies to large undertakings operating in the UK. The rules relating to the scheme are set out in the Energy Savings Opportunity Scheme...

Figure 1, CKAN interface

Publish and Manage Data

The web interface allows data publishers to easily upload and update their datasets in a distributed authorisation model called 'Organizations'. Each 'Organization' can manage its own access rights instead of a central admin. There is an organization admin page for managing members, datasets and dataset access.

Search Data

CKAN provides a rich search experience based on keywords, tags and browsing between related datasets. Other options like Fuzzy-matching and Faceted search are able to allow users to search for datasets without an exact keyword match. CKAN's search is powered by SOLR, an open source search platform.

Display Data

Data can be displayed by numerous previewing tools without downloading the data. This makes it easier for the user to inspect the data using an appropriate previewing tool for a certain file type.

Theming

Each CKAN instance can be themed to match the design of a certain organization. CKAN pages are generated from Jinja2 template files which can be modified. The design can then incorporate any messages or workflow that best suit the visitors.

Stakeholders

Making decisions regarding the CKAN project involves multiple stakeholders, each with different interests, requirements and needs. It is one of the most important tasks to correctly identify the stakeholders in order for the project to succeed, though this is often neglected in practice. In the following we discuss the different stakeholders based on Rozansky and Woods surrounding the CKAN project. First, a high-level diagram is shown that presents a simple overview. This is complemented by a more detailed description of each stakeholder and their influence.

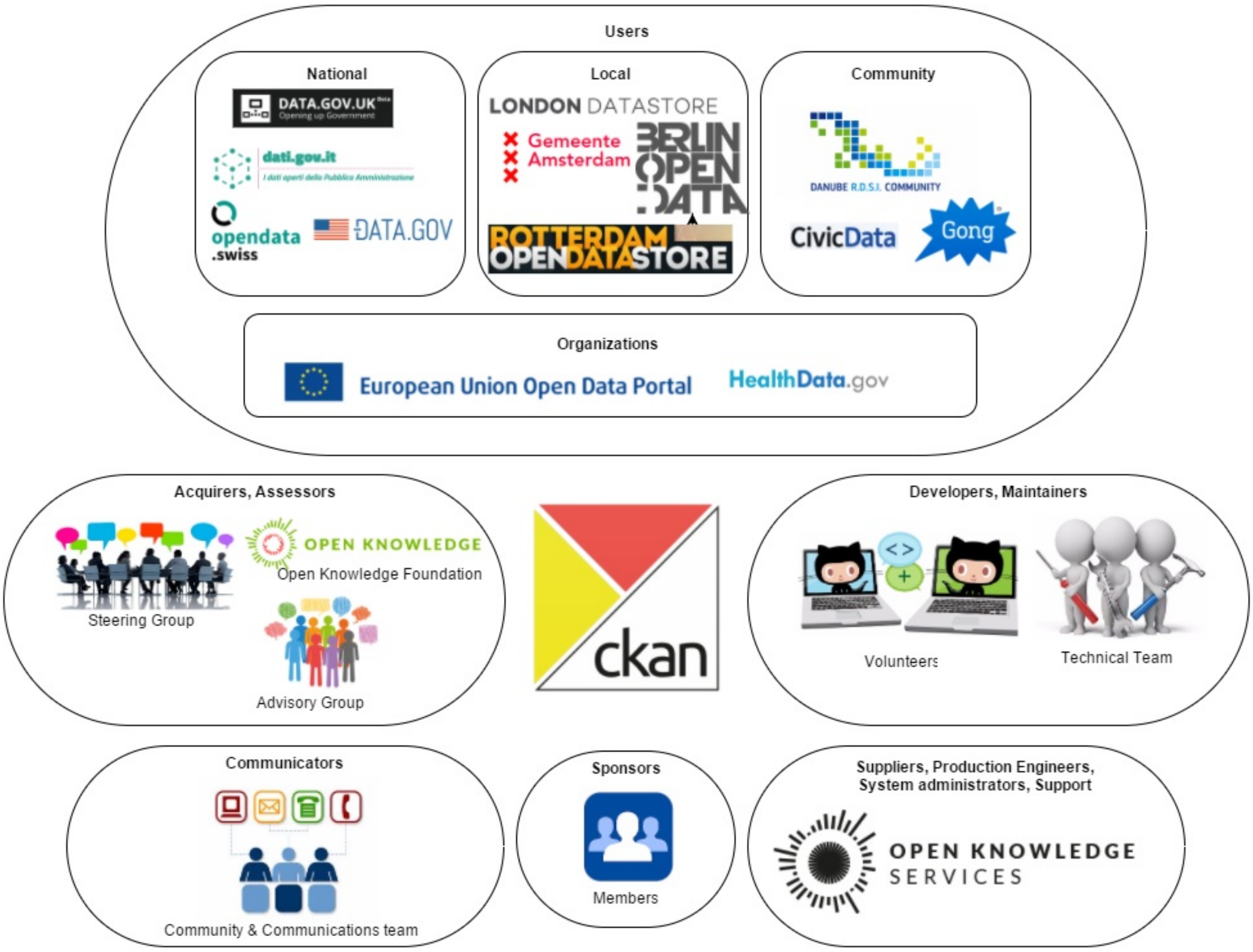


Figure 2, CKAN stakeholders

Stakeholder Description

The eventual **users** of CKAN are the governments, organization, and communities that wish to publish or collaborate with data. Their interests and needs lie in the usability and functionality of a data portal. They require control of data that gets published and is accessible most of the time. An extensive list of CKAN instances around the world can be found on their website. CKAN is open source and is free to download and install for any kind of use. The Open Knowledge Foundation has a professional team that provides paid CKAN services to assist in deployment, setup, hosting and support. This team belongs to Open Knowledge Services and can be categorized as stakeholders of the type **production engineers, system administrator, suppliers and support staff** with the Open Knowledge Foundation as their supplier.

The CKAN project started as an initiative of the Open Knowledge Foundation and is today overseen and managed by the CKAN association, independent and with its own governance. The steering group consists of key stakeholders who are committed to oversee and steer the CKAN association forward. Their main responsibilities are to oversee finance and activities within other groups including the legal and administrative aspects. The Steering group communicates with the advisory group which consists of members and representatives that support CKAN. The latter gives advice and expresses the needs of the community, but it does not have decision-making authority for the CKAN association. Both groups belong to the class **acquirers** and **assessors**. The community and communications team of the association are of the **communicators** stakeholder type. Its responsibility lies in informing key users, communities and the press trough meetings and online content with the purpose of increasing engagement. Finally, the technical team provides technical vision, makes major architectural decisions and has many more responsibilities regarding the software. Privileges like getting a vote on core changes and setting the technical direction of CKAN make it a very influential stakeholder. It is the core CKAN **developer** and **maintainer**. In addition, there are voluntary contributions to the code from around the world.

Apart from the stakeholders defined in Figure 2, we identify the **sponsors** which are the members of the CKAN association. Membership is a way for individuals, companies and organizations to support the CKAN Project and be recognized for doing so. They contribute resources by providing money and and-kind resources such as staff-time. The level of contribution varies between the different tiers of membership.

Power/Interest Grid

To achieve a better understanding of the different stakeholders, we present a power/interest grid. that provides a simple view of how influential each stakeholder is and the degree of interest. Identifying key stakeholders that are authorized to make decisions is essential for any architect.

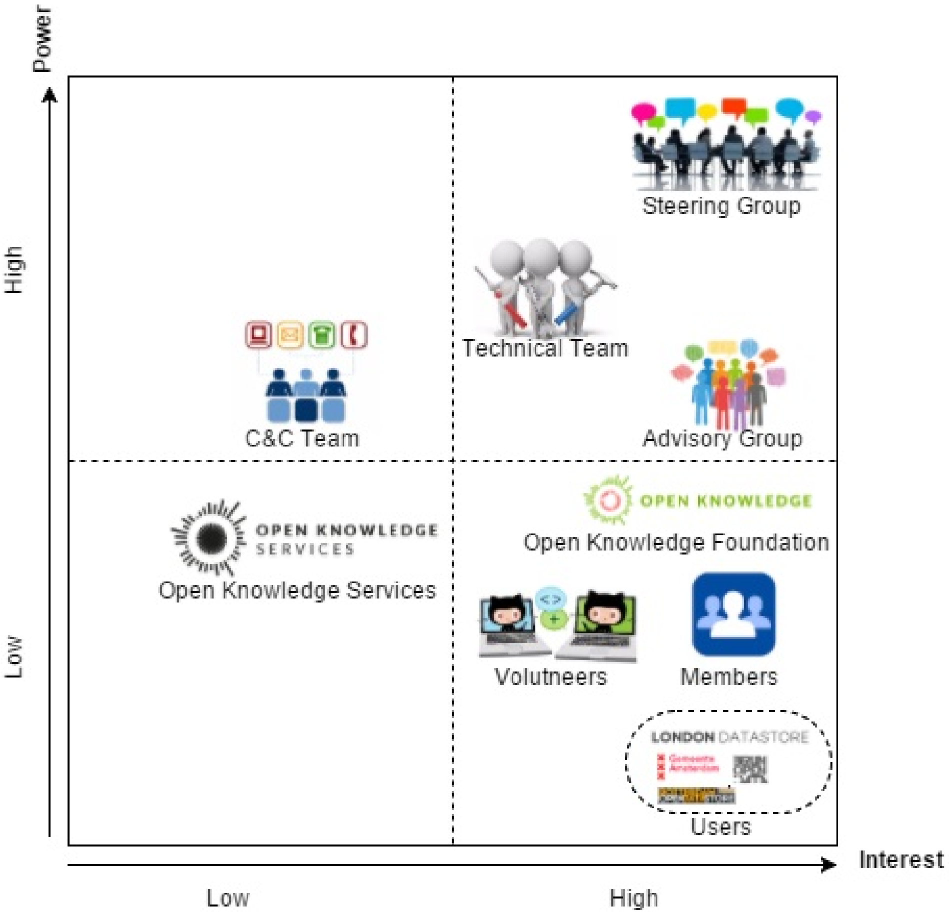


Figure 3, power-interest grid

From Figure 3 It is clear that the Steering Group has the most power that makes all major decisions, but it does receive input from the Advisory Group. The technical Team is in charge of software and the architecture with the privilege to vote on core changes.

Putting into context

With the service provided by CKAN, dataset owners are able to store and publish the raw data and metadata in well-structured and manageable ways. The users can then search for datasets of interest through both the website and APIs. CKAN also allows various extensions and plugins developed by third-party developers to enrich the functionalities of CKAN.

The context diagram presents the context of CKAN as well as the scenarios in which CKAN can be used. The CKAN Project is sponsored by the Advisory Group members who contribute resources, either through contributing money or providing in-kind resources such as staff time, for the development of CKAN. Some of the advisory group members, for example Open Knowledge, also provide technical support to other CKAN instances such as UK open data protal. The data published on CKAN instances can be used for research purposes, commercial exploitation or individual interests. The Technical Team takes charge of code contributions, technical documentation and all other technical-related issues using Github, testci and many other developing tools. Volunteer Developers are people who are interested in the project and willing to contribute to some coding area, for example fixing bugs and writing extensions for the CKAN platform. Finally, CKAN is not the only open data solution, competitors like FiscalNote and Socrata are playing the game as well.

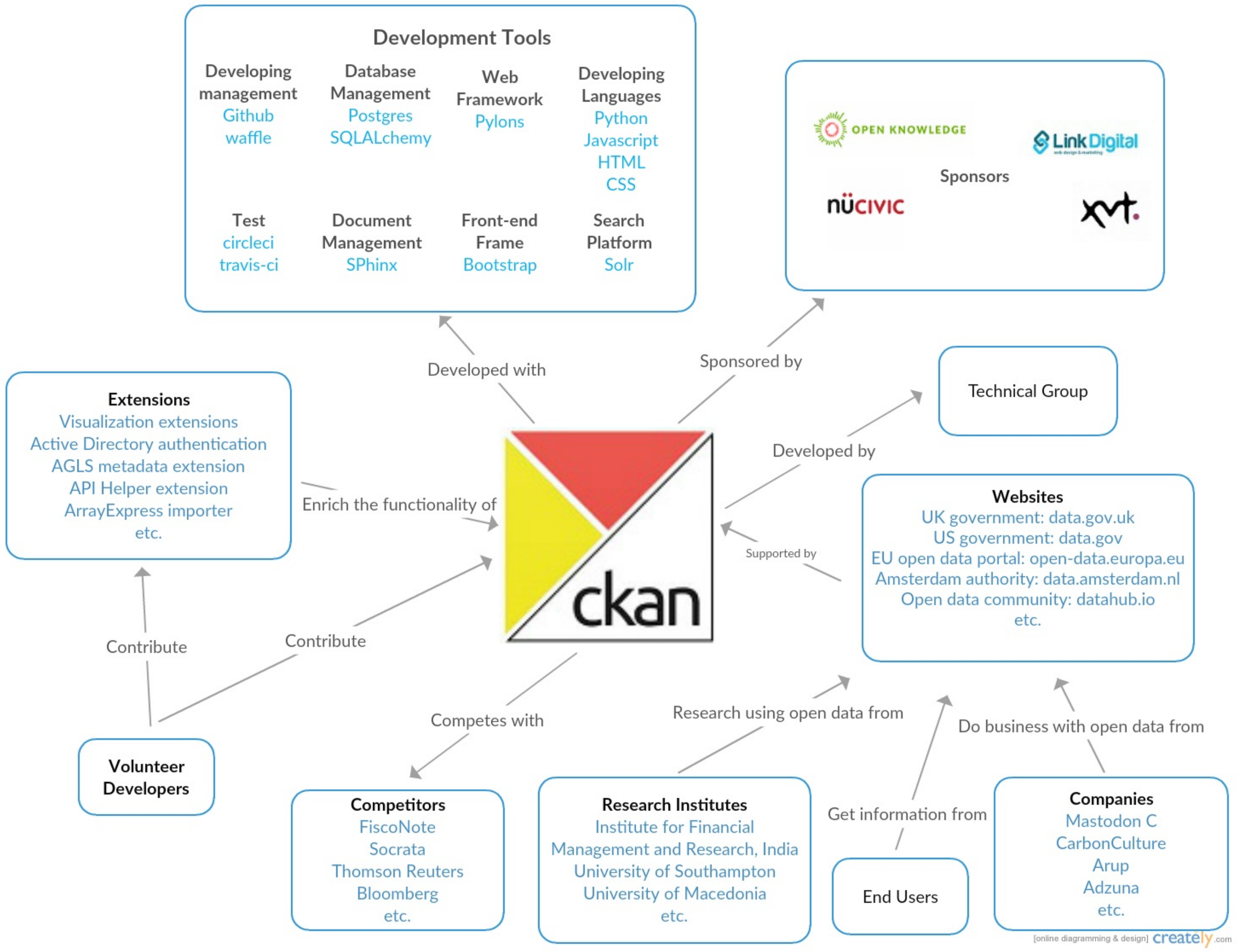


Figure 4, context view of ckan

Modules

We'll now have a look at the architecture of the CKAN project. These include the high level modules that provide CKAN's functionality and any architectural design that is system-wide implemented. Each module is categorized in a layer for which the dependencies between the different layers are shown.

Pylons

CKAN's internal structure is adapted to the structure of the Pylons Web Framework. This framework works a little differently than many other web frameworks. Normally, an entire framework is loaded first and then it searches for any project code to execute. Pylons does the opposite by importing objects while running the code, assembling a so called WSGI Application and returning it. This is done to achieve a higher degree of flexibility and customizability in building a web application. Pylons applies a Model-View-Controller (MVC) architectural design pattern. They have extended this slightly by not having the Controller directly interpret the client's request, it only acts to determine the appropriate way to assemble data from the model and render it with the correct template. In other words, the Controller calls portions of the model and view as necessary to fulfil the request.

Module Organization

In order to gain understanding of the architecture, we present a module structure diagram. Three different layers can be identified:

1. Controller layer
2. Model Layer
3. View Layer

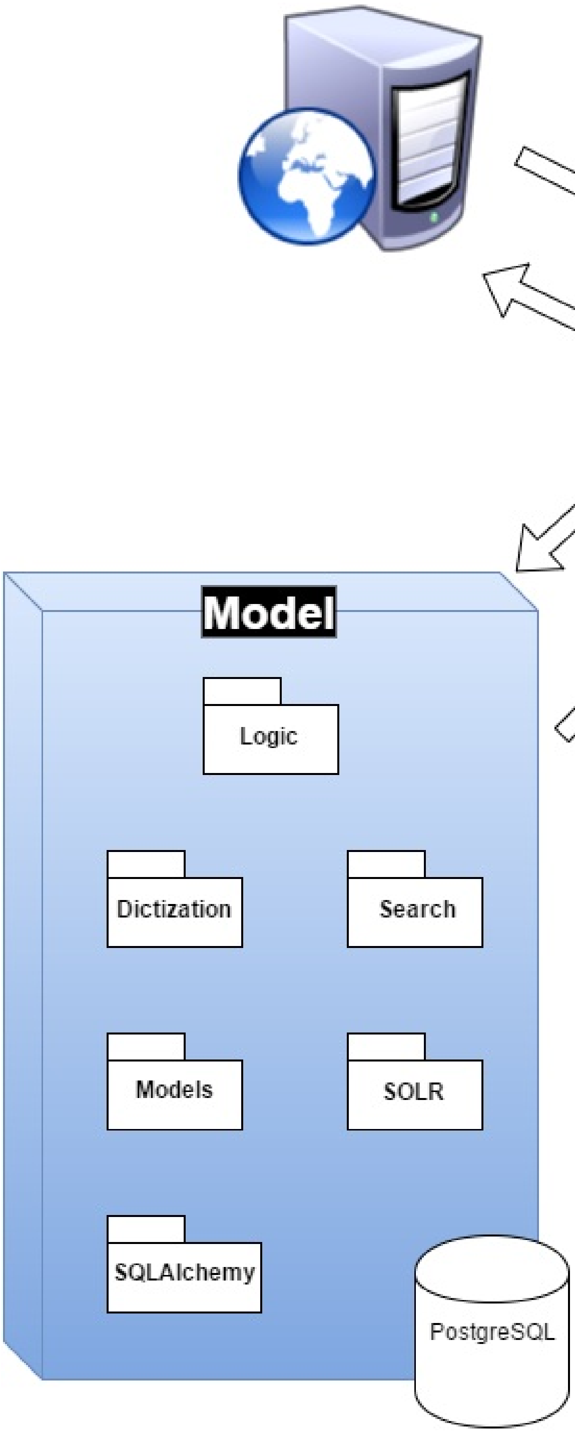


Figure 5, modules of CKAN

Controller Layer

When an HTTP request is being send by a client, the controller layer intercept this since this is how the Pylon framework handles requests. The purpose of the *Config* module is to apply middleware to the request. These can add additional functionality to the base application depending on the request. For example, the RoutesMiddleware parses the request URL to see if it has a matching controller, the information of the controller to be called is stored in the request. It then attempts to locate the class with a similar scheme in the *Controller* module.

Module Layer

The model contains all the necessary modules to perform non-trivial operations on datasets, organizations and users. The *Logic* module is most abstract which accepts calls from the controller to access data. Eventually, data are stored as atomic values in PostgreSQL, a relational database. This makes it harder to retain conceptual information that is present in objects when using an object-orientated programming language. SQLAlchemy comes to the rescue by taking care of the translation from objects to relational database entries and vice versa. For this purpose, the modules *Dictization* and *Models* are used. Search functionality is made possible by the *Search* module, which in turn will call the third-party SOLR library that will perform the actual search.

View Layer

The data retrieved from the model layer is display neutral, meaning that no formatting has been applied. CKAN uses a *Render* module which is responsible for generating templates with the requested data from the model. *Jinja2* is a designer-friendly template language that is imported by the render module.

Common Processing

Identifying common processing elements is necessary for any large system, not only because it helps avoid code duplication but also because it can improve the overall technical coherence and reduce the risk of falling into intricate programming traps. This is where a common design model is needed. A clearly defined common design model also makes it easier for the developers to understand, upgrade and maintain the system. In this section, we will provide an introduction to the common processes in CKAN and how it benefits from these design choices.

1. **Message Logging**

   CKAN uses the Python standard library's logging module to log messages. For different messages with different levels of destinations, the *logging* module supports writing to files, HTTP GET/POST, SMTP service, sockets or some OS-specific logging mechanisms. The log messages should be short and concise, accompanied with proper log-level (DEBUG, INFO, ERROR, WARNING or CRITICAL, see Python's Logging HOWTO). Well formulated logging messages help CKAN developers and maintainers trace bugs easily without wasting effort on identifying the code causing the problem.

2. **Internationalization**

   All user-facing strings in CKAN code are internationalized so that the translator could automatically localize the strings and translate them for various supported languages by CKAN. "Internationalization" here basically means passing the string to the translation platform, Transifex.

   CKAN uses urls to determine which language is used. For example, `/fr/dataset` will be shown in French. For now, CKAN already supports numerous languages. A list of supported languages can be seen here. The default language of a CKAN site can be simply changed by setting the *ckan.locale_default* option in the config file to desired language.

   CKAN's language support was contributed by a great number of volunteers, they work together in Transifex to create translations for CKAN. This resulted in the support of 62 different languages. The translation team includes 4 admins, 178 coordinators, 31 reviewers and 460 translators.

3. **Documenting the parameters, exceptions and returns of functions**

   CKAN requires developers to document the parameters and return values of functions. All the CKAN code should also properly document exceptions, including the type of exception and conditions. This informs other developers about the behavior of a particular function so that they know what to expect when using it.

4. **Third-party libraries**

   CKAN is developed with dozens of third-party libraries such as the Pylons Web Framework, the database ORM toolkit SQLAlchemy, the Bootstrap front-end framework, python standard library, etc. These libraries can be commonly used among different code modules consistently for solving one specific problem. For instance, Pylons can provide solutions to session encryption while SQLAlchemy contains encryption module for the database. The use of third-party libraries saves developers' efforts from solving trivial problems with existing solutions. However, these libraries might also limit the flexibility, leading to a tit-for-tat strategy.

5. **CKAN Library**

   In addition to the external libraries, CKAN has an internal library that contains:

   - Common Helper functions.

- Security: User Authentication and Security such as Captcha.
- Data Management: Data dictization, Package Search, Package data dumper, Resource saver etc.
- User Interface: Email Notification, Emails reader and sender, Date Formation, Webpage paginator etc.

These functions are isolated from the other modules to avoid circular dependencies and are easy to use by importing the corresponding modules. These functions help isolate common processes like dictizing data, dumping packages, and avoid code duplication.

6. **Logic functions**

   High-level functions like create, delete, search for and get, patch and update data from CKAN are available from the *logic* module in the Model Layer. Controllers are able to call them with `ckan.logic.action.get`. For each logic action, there is an authentication function to authorize the action. Besides, controllers can also validate external datasets in other formats and convert them to the proper format. These functions enable controllers to manage their actions in high level without considering how these actions are implemented in detail.

7. **Testing**

   All new modules of code or changes to existing code should have passed corresponding tests before being merged into master. The test modules in CKAN are maintained independently from the module being tested. They can be shared by many code modules. Therefore, developers can re-use the existing test modules instead of writing tests for every new change. We will also cover content about testing standards in testing standerds.

Codeline Organization

CKAN's source code structure, coding standard and testing standard are discussed in this section.

Source Code Structure

Figure 6 illustrates the general structure of CKAN. Folder *ckan* contains the key modules of CKAN, including models, views, controllers (the MVC pattern), data migration module, test module etc. Folder *ckanext* holds the user libraries. Other folders contain configuration files, bins or documentation.

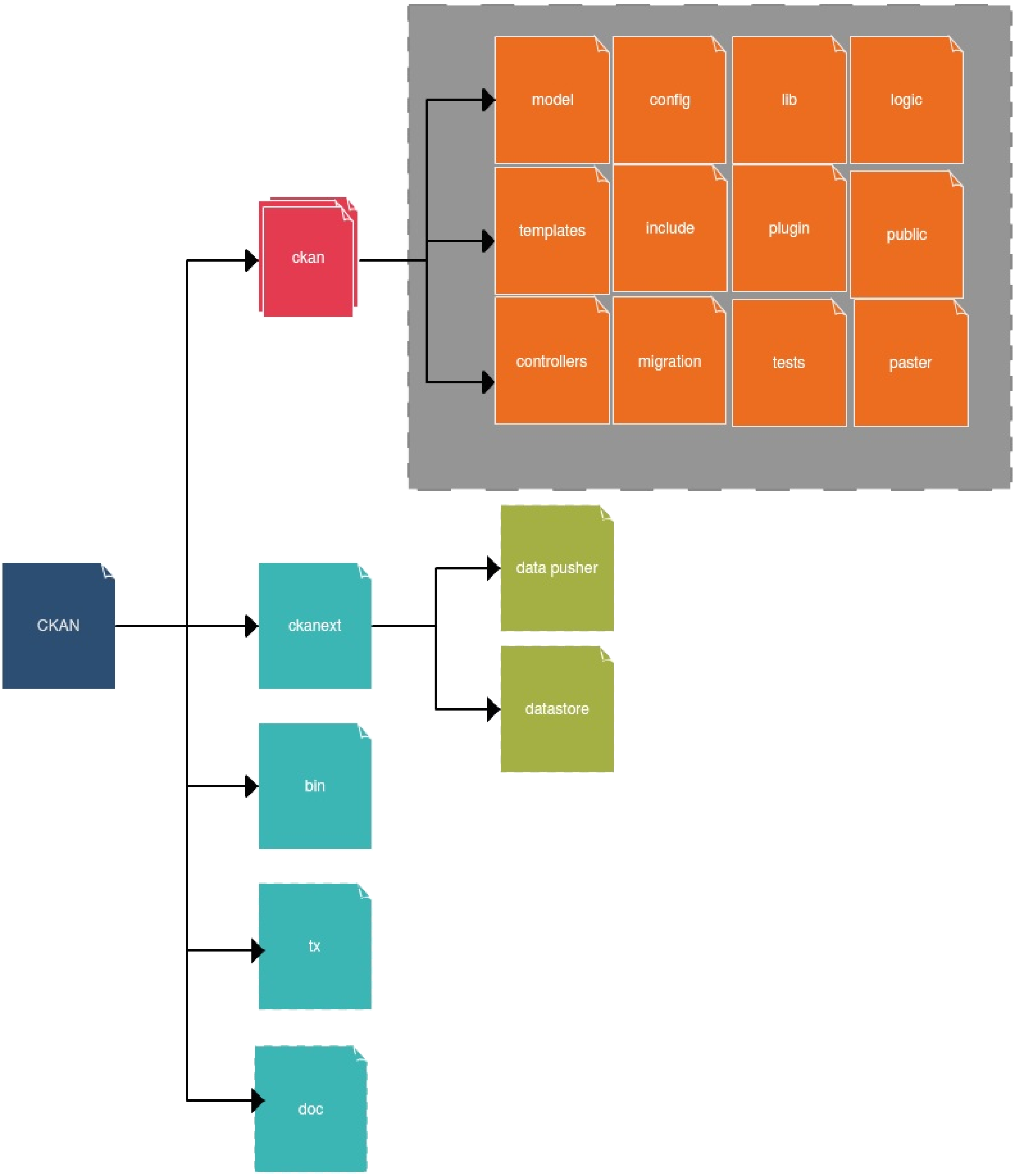


Figure 6, code structure of CKAN

Code Test Standardization

For CKAN, all new code or changes to existing code should have new or updated tests before being merged into master. Nowadays CKAN maintains 2 test styles named `ckan.test.legacy` and `ckan.tests`. Both the legacy tests and the new tests need to pass before merging into the master branch.

Release Process

The process of a new release starts with the creation of a new release branch. A release branch is the one that will be stabilized and eventually become the actual released version. *Beta releases* are branched of a certain point in master and will eventually become stable releases. Once the release branch has been thoroughly tested and is stable we can do a *final release*. A detailed release process can be found in Figure 7.

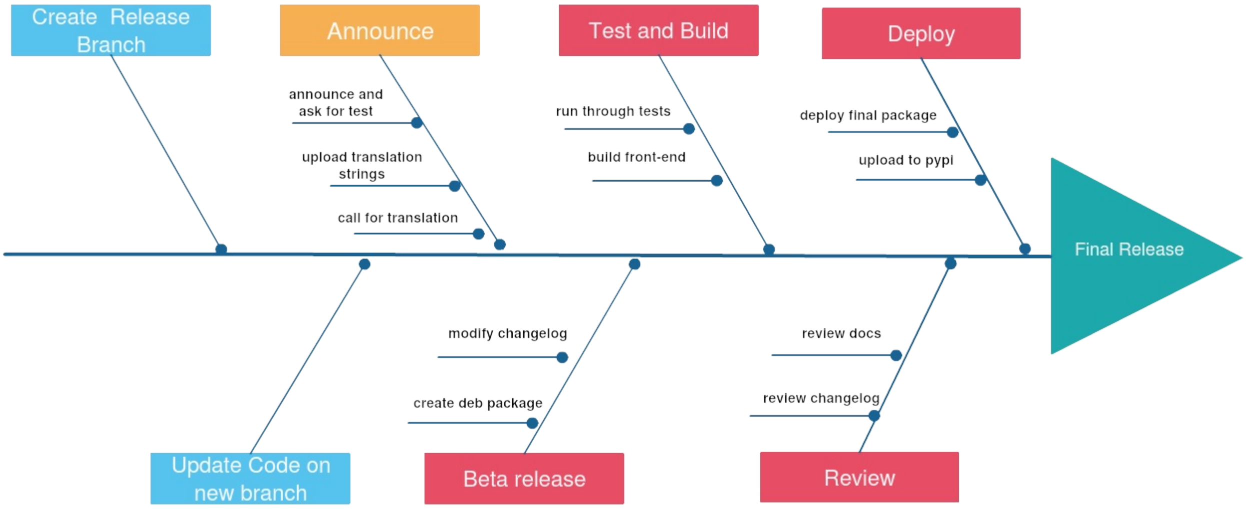


Figure 7, release process of CKAN

Information Viewpoint

Importance of an information system

An information system is defined as an organized system for the collection, organization, storage and communication of information. As we have already discussed in the previous chapters, CKAN is an open data management system. It allows data owners to streamline publishing, sharing, finding and using data. The essence of CKAN is an information system. Therefore, it could be interesting if we have a look at the information model of CKAN and how it satisfies the requirements of CKAN's features.

In order to get a taste of CKAN's information system, it could be helpful if we start with a brief overview of CKAN features regarding data management. Data nowadays usually come in large volume and is in various formats. The data management system should be able to store these data properly in an efficient and flexible manner. Besides, data needs to be easily accessed through web interface and APIs. Data should be indexed and searchable so that users are able to find data of interest conveniently. It might not be wise give everyone authorization to data, so fine-grained access control is needed in the system. In addition, data should be spatially and temporally consistent to users accessed from different locations or at different times.

Information Model

For these purposes, CKAN developed the following structures to manage scalability, flexibility, availability, consistency, persistency and safety of data:

Datasets and Resources

CKAN uses so-called "datasets" or "packages" as basic unit to store, publish and edit data parcels. Datasets can be temperature records from various weather stations or any other data. Each dataset consists of "resources" which is the data itself. In addition there is information about the data, referred to as "metadata". Resources can be stored as relational database entries or separate files (XML files, images, linked data in RDF format). Different storage metrics allow CKAN to manage data in various formats. For the structured data, each individual data elements are accessible and querible with SQL queries. For the unstructured data, files are atomic, meaning that there is no way to access or query parts of that file (see CKAN Documentation: Datasets and Resources).

The use of packages also improve the scalability of the information system. One can manipulate only the part of interest without considering the size of the whole data. Users can find the packages efficiently using provided search tools.

Different from other Cloud services like Netflix, CKAN does not use a distributed data store, because multi-user concurrency is not an important requirement for CKAN's information system. Instead, the integrity of data is usually favored by data publishers to avoid data conflicts. Therefore, CKAN decided to implement central data storage to satisfy their needs.

Users, Organizations and Authorization

As mentioned in Section Usage Perspective, CKAN users are structured based on "organizations", for example Environment Agency from the UK government. Users within organizations can create, edit and publish datasets depending on their authorization level which can be configured by the system administrators. When an action is taken, the parameters passed to CKAN are validated against a schema. The schema contains a list of functions that will validate the value of the corresponding parameters like the existence of datasets, the user permissions, etc. These schemas are customizable by the system administrator.

Data Flow

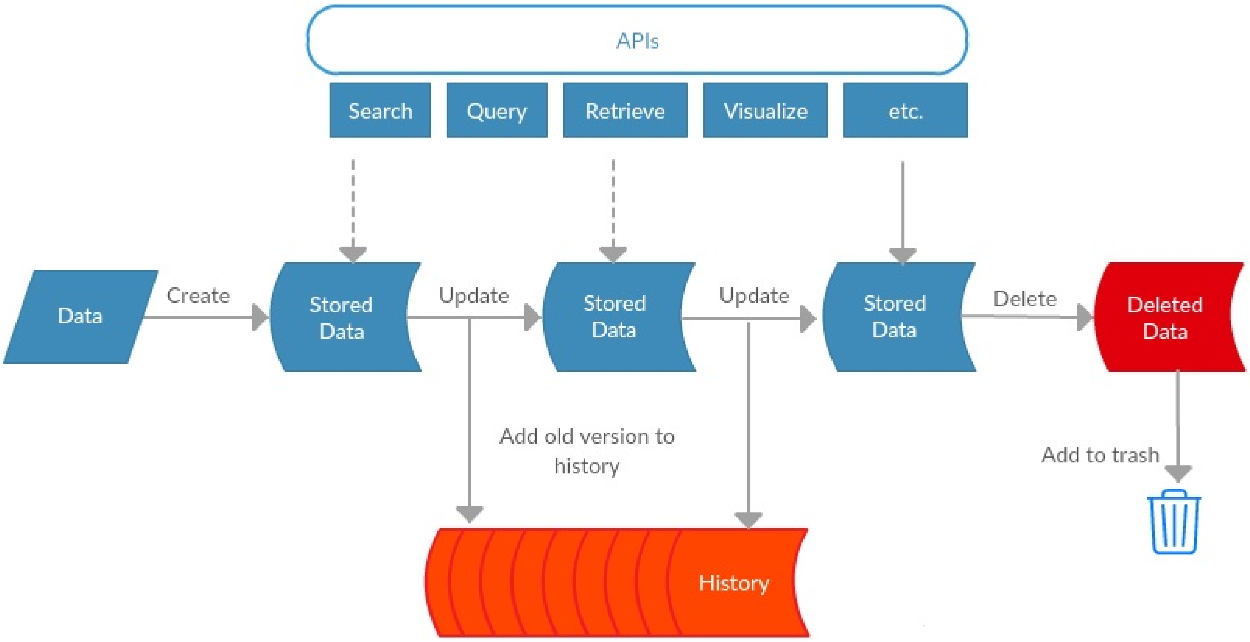


Figure 8. CKAN data flow

We have seen the static structure in the previous two subsections. In this part, we change our perspective to the dynamic flow in CKAN's information system. Figure 8 presents the life cycle of a dataset. It is first created (registered) by the users within an organization. Datasets can be updated during its life time. Every time it is updated, the old version gets added to the history of all edits and dataset metadata using the Open Knowledge Foundation's Versioned Domain Model (VDM). The latest dataset version is available for users before being deleted and added to trash. Datasets are usually accessed via various APIs. The web interface is also build on top of these APIs. Version control helps recover from corrupt data. It simplifies making changes and enhances collaboration among multiple editors within one organization.

Variability

Many of CKAN's features can be enabled by downloading the corresponding plugins and include them in a configuration file. These add additional functionality to the CKAN core. The configuration file also contains settings for toggling certain features, some are dependent on one another. For example, specifying a default resource view requires that the corresponding plugin is loaded. Also, if datasets don't have to belong to an organization then creating unorganized users must also be enabled. A conflict between features can arise if both simple and faceted search is enabled. Finally, a language must be specified in the *Locales Offered* setting before it can be set as *Locale Default* or included in *Local Order*. A diagram from FeatureIDE is shown in Figure 9 that visualizes some features and their dependencies. To keep the size manageable, only a selection of features are included that have some dependency.

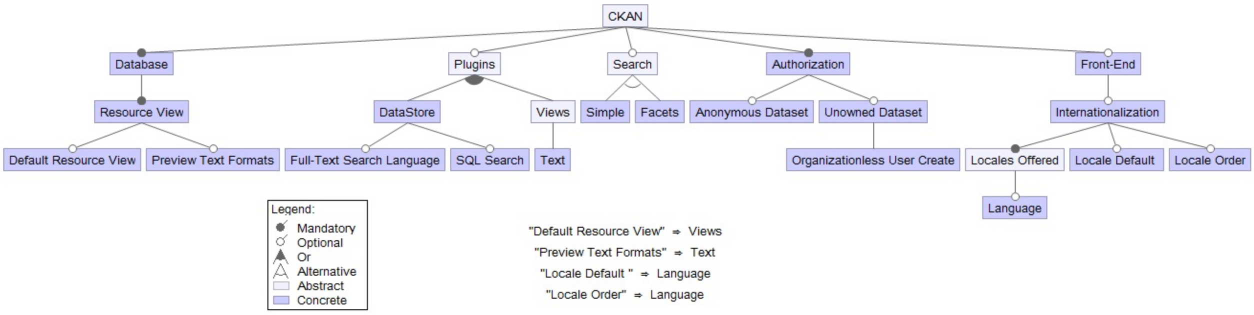


Figure 9, CKAN feature IDE

These configurable features can be changed at different binding times. we classify these features into two categories: boot time features and runtime features.

Boot-Time Features

CKAN configuration options are generally defined before starting the web application (boot-time), these options can be modified via configuration files or environment variables.

These are generally low-level critical settings needed when setting up the application, like the database connection, the SOLR server url, etc. Sometimes it can be useful to define them as environment variables to automate and orchestrate deployments without having to first modify the configuration file. These options are only read at startup time to update the *config* object used by CKAN, but they won't be accessed any more during the lifetime of the application.

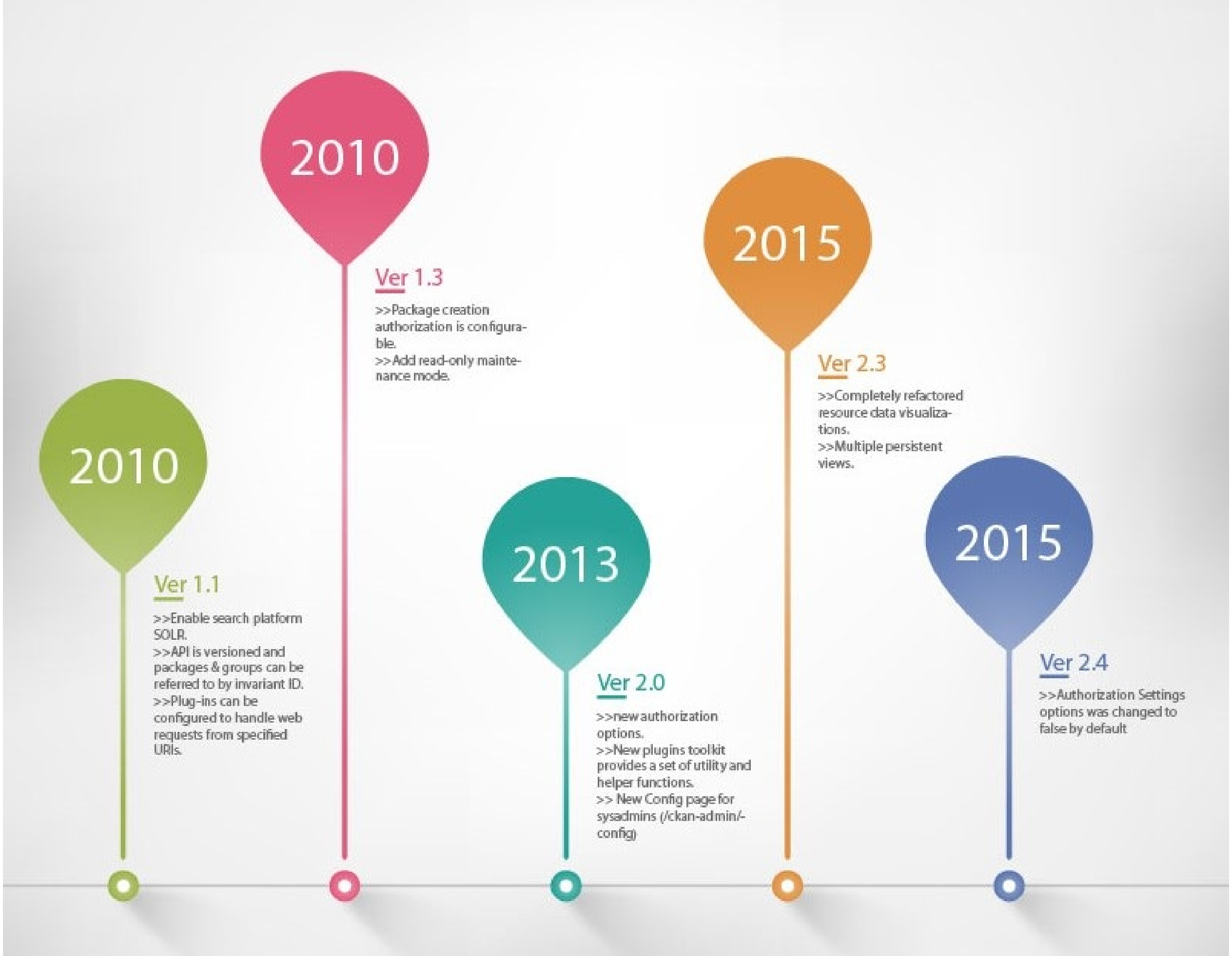
Run-Time Features

A limited number of configuration options can also be edited during runtime. This can be done on the administration interface or using the `config_option_update()` API action. Only *sysadmins* can edit these runtime-editable configuration options. Changes made to these configuration options will be stored in the database and persists when the server is restarted. In addition, Extensions can add (or remove) configuration options to the ones that can be edited at runtime.

Configurable Feature Evolution

CKAN has already been developed for years and open-sourced from the very beginning. A full changelog with detailed descriptions starting from 2009 (v0.10) is available for the evolution history and a ideas and roadmap is set for the direction of CKAN's evolution.

These help a lot for our analysis of the evolution of CKAN's variability mechanisms and features. In Figure 10 we will highlight some of the most important configurable features as examples of the evolution.



Ver 1.1
>>Enable search platform SOLR.
>>API is versioned and packages & groups can be referred to by invariant ID.
>>Plug-ins can be configured to handle web requests from specified URIs.

Ver 1.3
>>Package creation authorization is configurable.
>>Add read-only maintenance mode.

Ver 2.0
>>new authorization options.
>>New plugins tool kit provides a set of utility and helper functions.
>> New Config page for sysadmins (/ckan-admin/-config)

Ver 2.3
>>Completely refactored resource data visualizations.
>>Multiple persistent views.

Ver 2.4
>>Authorization Settings options was changed to false by default

Figure 10, several example configurable features evolution of CKAN, for example, search configuration, API configuration, authorization configuration and sysadmins configuration

Technical Debt

Technical debt is a metaphor referring to the eventual consequences of any system design, software architecture or software development within a codebase. Technical debt is not necessarily a bad thing, and sometimes it is required to move projects forward. But if technical debt is not repaid, it could continue accumulating, making the system hard to maintain and implement changes later on.

Coding Language and Test Suite Update

One of the biggest consequences of technical debt CKAN now faces is to maintain and update the old code base. Since CKAN uses many different development tools and coding languages, it has to keep the code up to date to accomodate the latest version of each tool. Issue#2772 is an example that CKAN wants to update its CSS pre-processor. These kinds of updates usually mean rewriting a lot of code and may cause some compatibility issues.

Lack of Documentation

This open source project was build and developed by multiple developers. Documentation is an essential part for understanding each other's work. CKAN has created a *doc* folder to store all documents. But this folder is not well structured and can be hard for user to find relevant information. Also, most of the files lack description about their usage.

Defects

While developing, some pieces of code may not work as intended. Bugs are inevitable even with a strict test process. The accumulation of defects may cause technical debt. There are a lot of issues that report defects and some of them remain unsolved.

Figure 11 illustrates the analysis result returned by SonarQube. The majority of CKAN's technical debt is focused on efficiency and maintainability. These kinds of technical debt are mainly caused by non-standard code. SonarQube gave CKAN a score of *A* and the technical debt ratio was 1.0%. Results indicate that code duplications account for 2.6% of CKAN files, which exists in 108 blocks and 23 files.

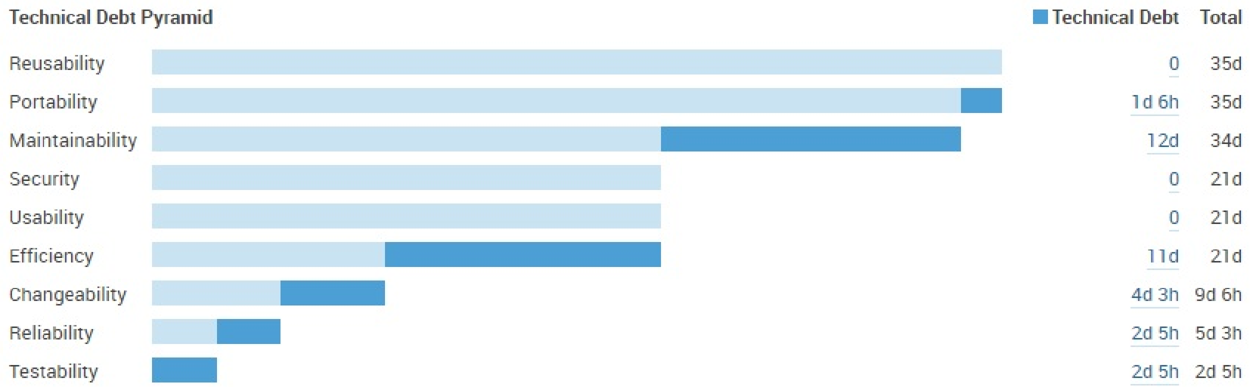


Figure 11, technical debt of CKAN

With the help of the ideas and roadmap, the CKAN team is able to identify the features, functionalities, services or apps to be discussed. Estimation of effort can be made for each of the issues. The development schedule is then made based on the discussions. This mechanism of development management can help the developers identify the technical problems in advance and reasonably allocate their effort so that unnecessary technical debts can be avoided. In addition, developers will also benefit from this clear development roadmap when they have to make an inevitable technical debt by adding the debt to the Backlog and coming back to it when it gains higher priority.

Conclusion

In this chapter, we provided an overall view for an open source data management portal named CKAN together with various detailed perspectives.

The key features of CKAN are publishing data and making them easily searchable by anyone that is interested. Since its release, many governments, organizations and communities have launched a data portal powered by CKAN. This can be considered a success and a major step towards realizing the idea of Open Data. The steering group, advisory group and techincal team are the most important stakeholders. They will keep steering CKAN in the right direction. CKAN's internal structure resembles an MVC-model where functionality is clearly separated from the interface. Common processing and codeline organization help the development team to remain consistent. CKAN developers made a lot of effort to make sure that each CKAN instance has customizable configuration options to satisfy their user's needs. Still, there is not much dependency between the different features. Since its birth in 2010, CKAN has evolved to a mature project where improved authorization and support for multiple data views are notable changes. However, changes came at a cost and have introduced technical debt primarily in maintainance and efficiency due to non-standard coding.

We are grateful for having the chance to study a well-developed open source project. In this process, we have learned a lot about the exterior and interior of CKAN. Stakeholders play an important part and it is key to consider their needs throughout the development process and evolution of the system.

CodeCombat: learn how to code by playing a game.

Maikel Langezaal, **Yu Liang**, **Chengxin Ma** and **Martijn Cligge**.

Delft University of Technology, 2016

Abstract

CodeCombat is an online multiplayer game designed for users to learn how to code. This chapter gives an overview of the software architecture of CodeCombat by adopting multiple views and perspectives. The system is analyzed from shallow to deep. It starts off with some more high-level analyzes, like a feature and stakeholder analysis. After that, a more technical analysis of the architecture has been made, like a development view and a variability analysis. One of the findings of this analysis is that the system is well organized but lacks some standardization. Lastly, the technical debt of the system has been analyzed. This presented some aging libraries and inadequate documentation. Overall, CodeCombat is a structured system with many features and it is continuously being improved by many stakeholders.

Table of content

1. Introduction

Coding is becoming more and more important in the modern day world. Some people even say that coding should be a mandatory class in school and most companies nowadays require their employees to have some basic coding skills. Coding can be really useful in many situations, for example building an innovative application or solving a complex mathematical problem.

So, imagine that you are inspired by all the things you could do with coding, which options are available to learn how to code? You could start by reading coding books and doing boring exercises, ending up scratching the head and staying up late night after night. But if you are interested in a more fun and interactive way of learning, CodeCombat might be an option. CodeCombat is an online multiplayer game where people can learn how to code by playing. The game is set in the dark ages and the user plays the role of a knight. CodeCombat is created with a mission; it intends to be an integral part of the programming community where newcomers can learn how coding can be fun and programmers of higher skill level can play to challenge and expand their skills. CodeCombat was founded in 2013 by Nick Winter and has more than five million active players. The source code of the system is publicly available via its repository on Github.

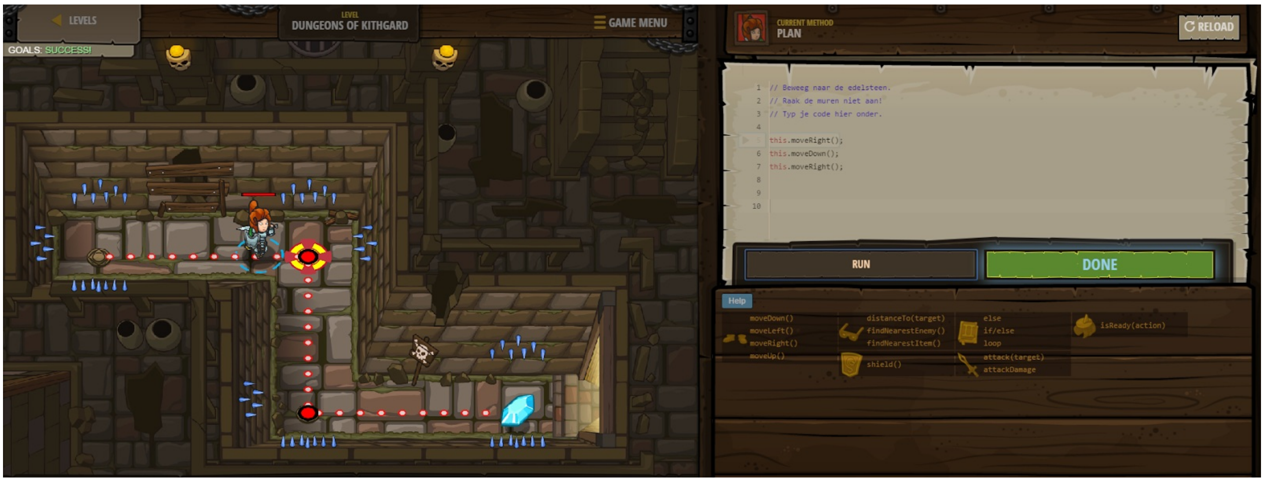
Four TU Delft students from the DESOSA (Delft Students on Software Architecture) group have made an in-depth analysis of the architecture of the CodeCombat system. The analysis is based on different architectural views and perspectives. The chapter starts with a feature description, followed by a functional view. After that, a stakeholder analysis will show who is involved in the development of the game. Subsequently, a context view has been made to discover the interactions between the system and its environment. The development view will elaborate more on the modular structure of the system while the usability perspective shows how the user interacts with the system. The variability perspective will show how variability is handled. Lastly, the technical debt analysis will describe how well the system is implemented.

2. What is it CodeCombat? - *The start of your coding adventure*

CodeCombat is a feature rich web-based game and is compatible with most mainstream browsers, such as Chrome, Firefox, Safari, and Internet Explorer. The game is available in multiple natural languages and is therefore easily accessible for international players. The user can learn six different programming languages by playing the game (Javascript, Python, CoffeeScript, Clojure, Lua, and Io). To get started, the first thing a user has to do is create an account. The user can log into his personal account via its Google, Facebook, or CodeCombat account. There are different type of accounts available; there is an account type made for teachers, one for students and one for normal players. The students can choose of 6 different courses which can teach them specific programming skills, and teachers can manage these courses via their account.

Now the coding journey can start! The player can either start playing a level created by him-/herself (Yes! It is possible to create levels by the players themselves!) or choose one from the campaign levels. After the level selection, a user has to choose a game character and attach items to this character, like a sword or some type of clothes.

But how do these levels look like? In the screenshot in figure 1 the user interface of the game is shown. The game character can be seen on the left, and is controlled with the code editor on the right. An example of this is command `this.moveRight()` which moves a game character to the right. In-game settings (like the sound volume, zoom in/out, and full-screen mode) and code editor settings (like enabling autocompletion of codes) can be configured by the user.



```
// Beweeg naar de edelsteen.
// Raak de muren niet aan!
// Typ je code hier onder.

this.moveRight();
this.moveDown();
this.moveRight();
```

```
moveDown()
moveLeft()
moveRight()
moveUp()
distanceTo(target)
findNearestEnemy()
findNearestItem()
shield()
else
if/else
loop
attack(target)
attackDamage
isReady(action)
```

Figure 1 A screenshot of the game

Besides playing the regular game, CodeCombat offers users the option to create their own levels and game characters. This is done by two features: the level editor and the thang editor.

Users can create their own level via the level editor. The level editor is based on the Thang Component System. Different items (like barrels, a background, and walls) can be added to the level via the level design section. Additionally, scripts (defining what is happening in the level, what the goals of the level are and which programming language the user can learn), sample codes (coding hints for the code editor), so-called systems (defines how the level renders), and some other small details (like a name for the level) can be added to the level.

To motivate user's creativity, game characters, and items (both called Thangs) can be customized via the Thang editor. The behavior and the characteristics of a game character (for example if a character can jump, or have certain fight strength) can be modified in the components section. Users can change the name of their Thang and edit the general configuration of their Thang like the color, type of body parts, sounds it makes and animations of movement. Lastly, users can choose to modify or create items, like a sword or shoes. These items can be used later while playing the game.

3. Functional view - *What functions does the system perform?*

All of the previously described features perform certain functions. The functional view of CodeCombat demonstrates how the system performs these functions. It is expressed in the fashion of a Boxes-and-Lines Diagram.

The diagram in figure 2 shows the functional view of the system with functional elements. A functional element is a software code module, an application package, a data store, or even a complete system. Each functional element holds some responsibilities. The functional elements are derived from the features of CodeCombat, as seen in Table 1. In CodeCombat, a functionality can be as concrete as "enlarging the editor size by a factor of 150%", or as abstract as "editing self-defined levels". To make the model simple and clear, functional elements in the diagram stay at an abstract level.

Functional elements	Feature
User Information System	• Log in • User types • Account settings
UI Language Selector	• Internationalization
Course System	• Courses
Programming Language Selector	• Programming languages
Level Editor	• Level editor-Level design • Level editor-Level script • Level editor-Sample code • Level editor-documentation • Level editor-Systems
Thang Editor	• Thang editor-Components • Thang editor-General Configurations • Thang editor-Item Editor
Multiplayer Mode	• Multiplayer levels
Game Play Core	• Game version • Game characters • Game items • Game campaign levels
In-Game Tweak Tool	• General game settings • Editor settings

Table 1 Functional elements and their features.

In the center of the diagram the core part of the game is shown, i.e. the game play function. This function depends on the programming language selection, user interface language selection, (optional) use of editors, in-game tweak tool, and user information system. Also other functions, including the course system and the multiplayer mode, depend on this game play core. The direction of the arrows indicates the dependency among the function elements.

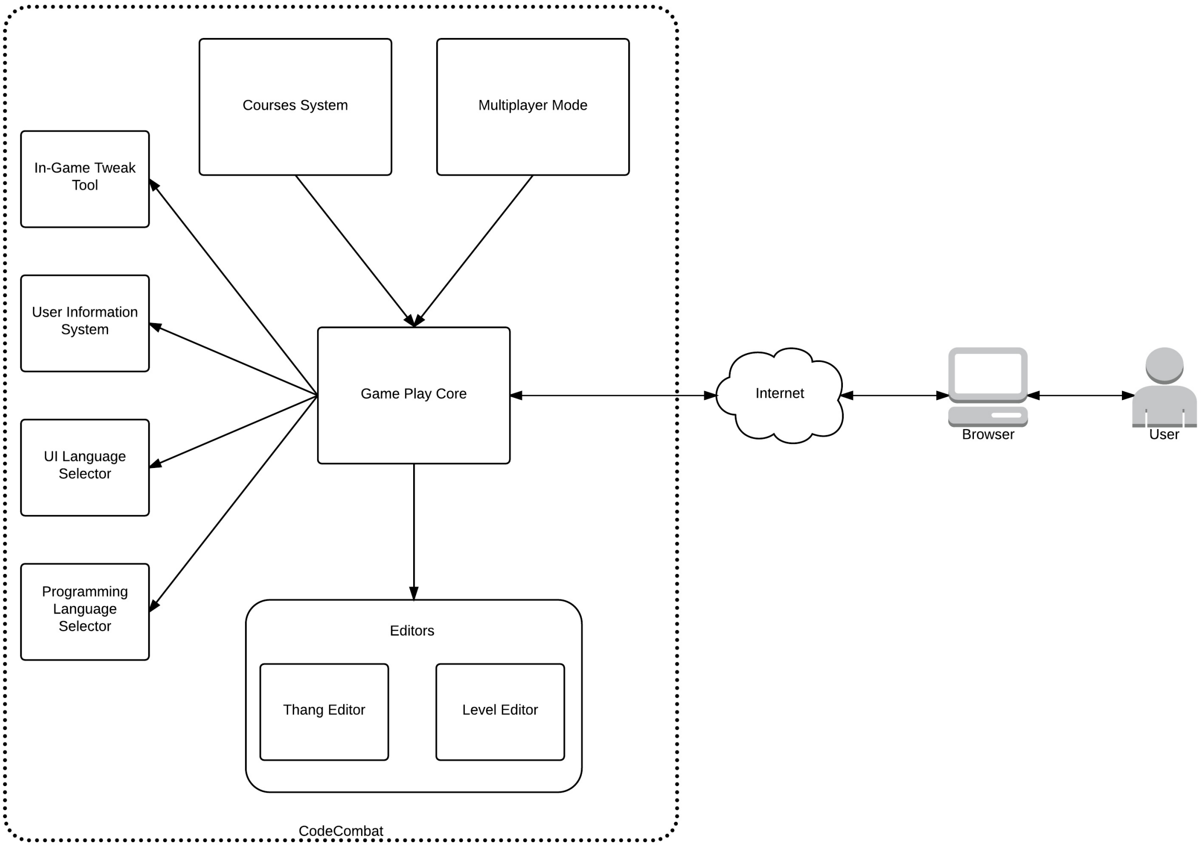


Figure 2 CodeCombat functional view diagram

This brief overview showed that CodeCombat is a very feature-rich game with many different functions. Developing a game with many different features requires a diverse development team. In the following section, the development team of the game will be discussed.

4. Who is involved? - *The builders of the empire*

4.1 Stakeholder analysis

To get to know the people who are responsible for the CodeCombat project, a stakeholder analysis has been made. The book *Software architecture of Rozanki & Woods (2012)* provides a handy overview to identify different type of stakeholders. The stakeholders are identified by looking at pull requests and issues on their Github page, and by looking at the website of the game. The complete set of stakeholders is shown in figure 3. Some of the most important are discussed below.

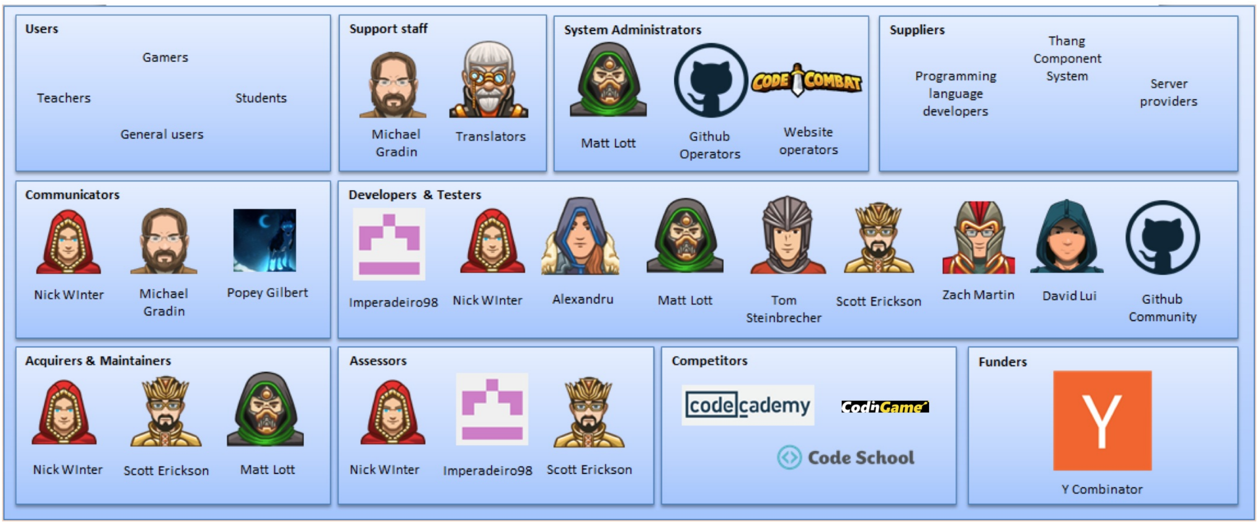
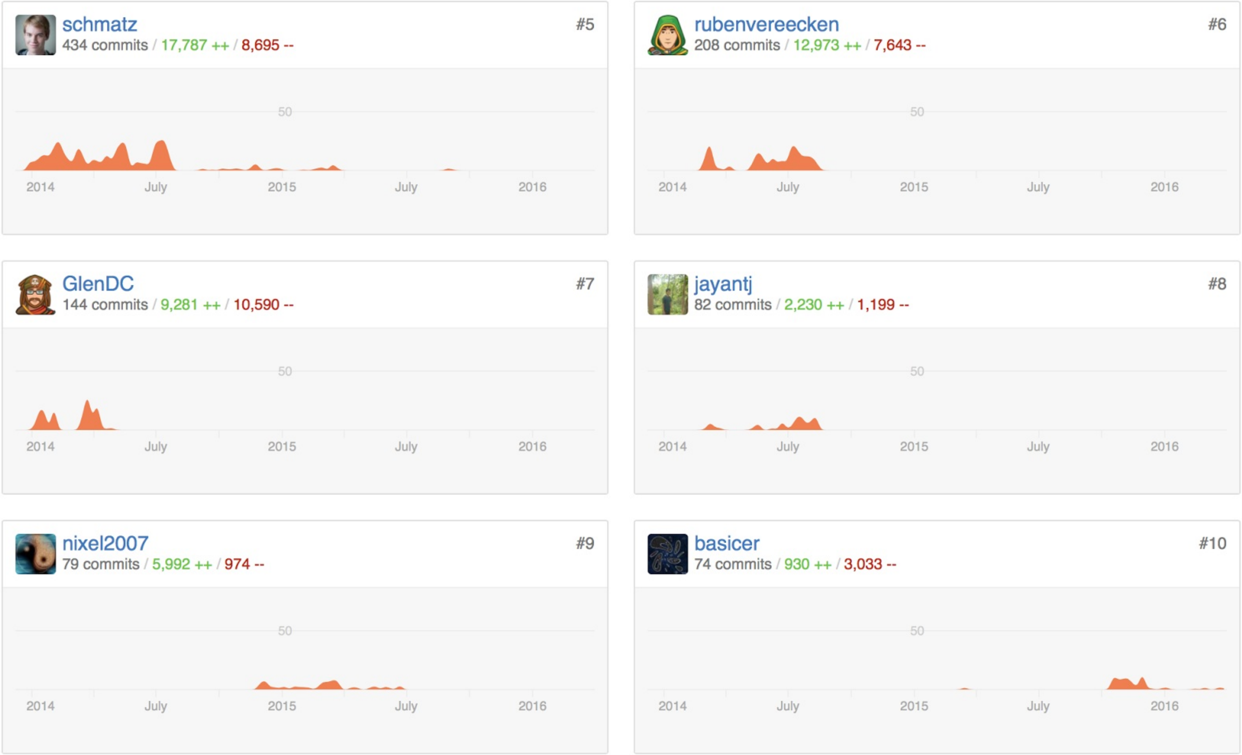


Figure 3 Complete overview of stakeholders

Developers and testers

Developers and testers are the ones who develop a product/system from specifications and then test it. Their main activities include coding and testing.

The pull request and issue analysis shows that Imperadeiro98, nwinter, differentmatt and Scott Erickson are the most active developers and testers on the repository of CodeCombat. They made and merged most of the pull requests and raised most of the issues. But, there are many more developers and testers can be found on the Github contributors tab and their website (see figure 4 for Github users, and figure 3 for developers on website).


schmatz #5
434 commits / 17,787 ++ / 8,695 --

rubenvereecken #6
208 commits / 12,973 ++ / 7,643 --

GlenDC #7
144 commits / 9,281 ++ / 10,590 --

jayantj #8
82 commits / 2,230 ++ / 1,199 --

nixel2007 #9
79 commits / 5,992 ++ / 974 --

basicer #10
74 commits / 930 ++ / 3,033 --

Figure 4 Github contributors

Acquirers, accessors, and maintainers

Acquirers oversee the procurement of the system while maintainers oversee the evolution of the entire system once it is operational. Accessors oversee the system's coherence on standards and legal regulations.

Firstly, the acquirers are Nick Winter (CEO) and Scott Erickson (CTO), the founders of CodeCombat. Secondly, the pull request and issue analysis showed that the both of them are also taking care of most of the technical issues and the overall evolution of the system, so they can also be identified as the maintainers of the system. Lastly, the assessors are Nick Winter, Scott Erickson, and Github user Imperadeiro98. Nick Winter focuses more on the intellectual property issues, which are a key factor in developing software projects to avoid getting involved into legal problems while Scott Erickson and Imperadeiro98 have the responsibility to check if commits on Github comply with the standards used in the CodeCombat repository.

The communicators and supporting staff

Since CodeCombat is an open source project, communication with the community is very important. Communicators provide documentation while the supporting staff provides support to the users.

On Github, Popey Gilbert edits the Wiki page of the CodeCombat repository and is therefore the main communicator. CodeCombat provides additional documents about other information on their website and blog. The website and the blog function both as documentation-and support tool. Nick Winter is posting messages on this blog and Michael Gradin updates their website. So, Nick and Michael can be identified both as support staff, as well as communicators.

Another kind of support staff are the translators, whose responsibility is to translate the game into other natural languages. These translations are being made most by a variety of Github users.

Users

The user group mainly consists of students who want to learn to code, teachers who use it as educational material, and general users who just want to play a game.

Others

Other stakeholders include the Funder of the project, Y combinator. Competitors like Code School and Codingame, system administrators like Matt Iott, and suppliers like server providers.

4.2 Power-interest matrix

A rich set of stakeholders is the result of the stakeholder analysis. To get a more structured overview of the set of stakeholders with their power, a power interest matrix has been made (see in figure 5).

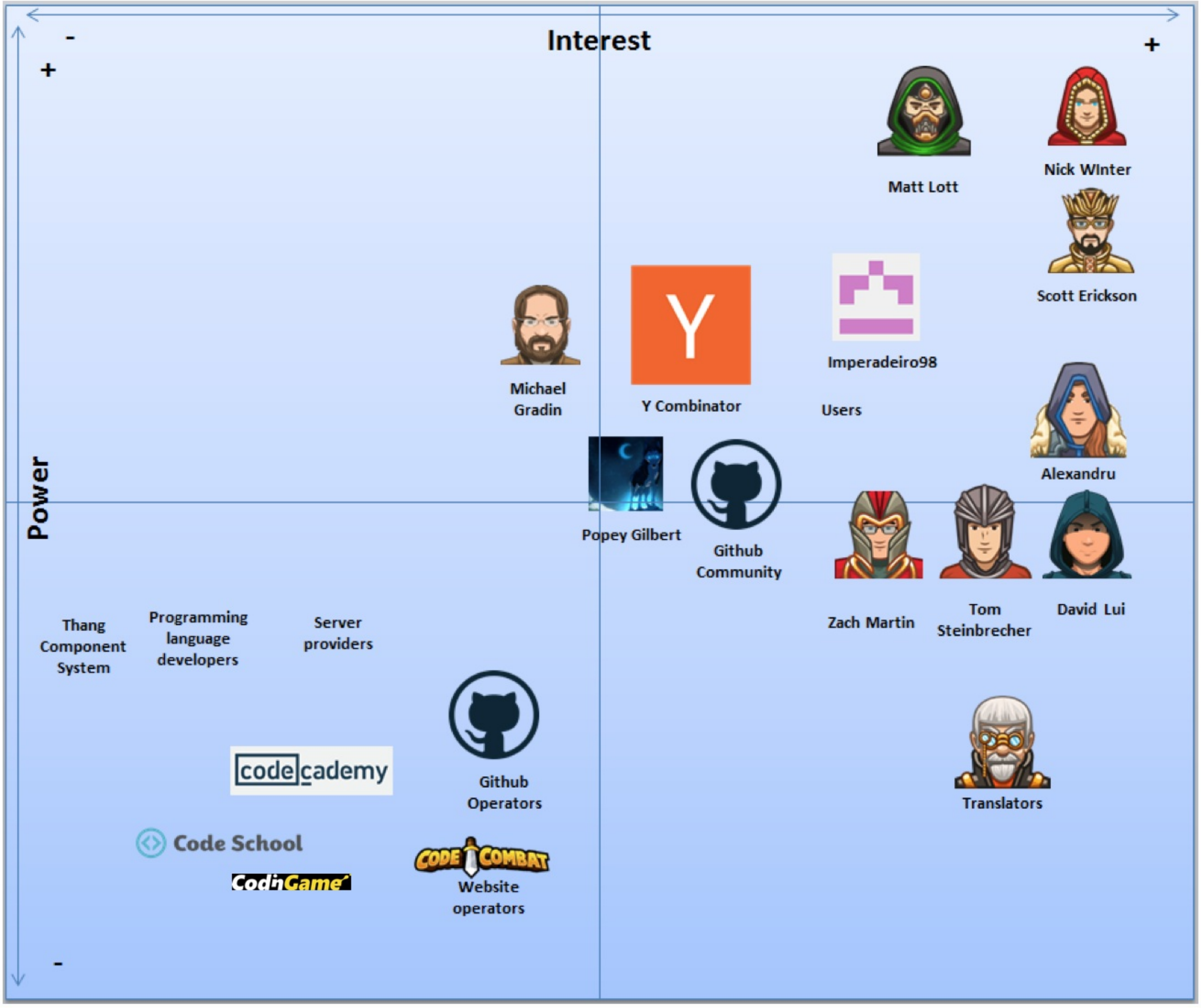


Figure 5 Power-interest matrix

The stakeholder analysis showed that Scott Erickson, Nick Winter are the most important people who are responsible for the development of the game. To get to know the environment of the game and the interactions between CodeCombat and its environment, a context view has been made (again based on *Rozanki & Woods (2012)*).

5. Context view - *The relationships between CodeCombat and its environment*

Now it is time for a more technical analysis of the system! The context view of a system defines the relationship between the system and its environment.

5.1 System scope and responsibilities

System scope and responsibilities of CodeCombat define what the system should do in order to fulfill its objective, which in this case is providing users an environment where they can learn to code by playing a game. As mentioned in the section of functional view, the scope and responsibilities of the system include the following:

- Enabling the users to register, login, and edit their personal accounts
- Enabling users (specifically teachers) to create classes or users (specifically students) to join classes
- Providing a forum where users can discuss the game
- Providing game related aspects (like game level selecting, game role choosing, and game coding language choosing)
- Providing different language options (Both natural languages and programming languages)

5.2 External entities

There are ten external entities in the CodeCombat system, and together they make up the environment where CodeCombat resides. The ten external entities are listed below:

- Supported browsers: mainstream browsers like Chrome, Firefox, Internet Explorer, and Safari
- Developing language: CoffeeScript, Jade, Sass, and HTML
- Server side support: the backend support of the system, including system server (Node.js), database used to store the data (mongoDB), the web framework (Express.js), and libraries used on the server. More details about the libraries can be found here: Third party software and services
- Browser side support: third party software like sitewide libraries, gameplay libraries, and services (e.g. Box2D, a physics engine). More details about the libraries can be found here: Third party software and services
- System testing tools: Karma, Jasmine, and BrowserStack
- Developing and maintaining platform: Github
- Continuous integration tool: Travis CI (which ensures the testing of every pull request on the Github before merging)

- Communication tools: Discourse (a forum software) and SETT (a high-engagement blogging software)
- License support: MIT license

5.3 Context view diagram

The diagram below in figure 6 shows the context view of CodeCombat. The context view diagram shows the system scope and responsibilities, external entities, as well as the most important stakeholders of section 4.

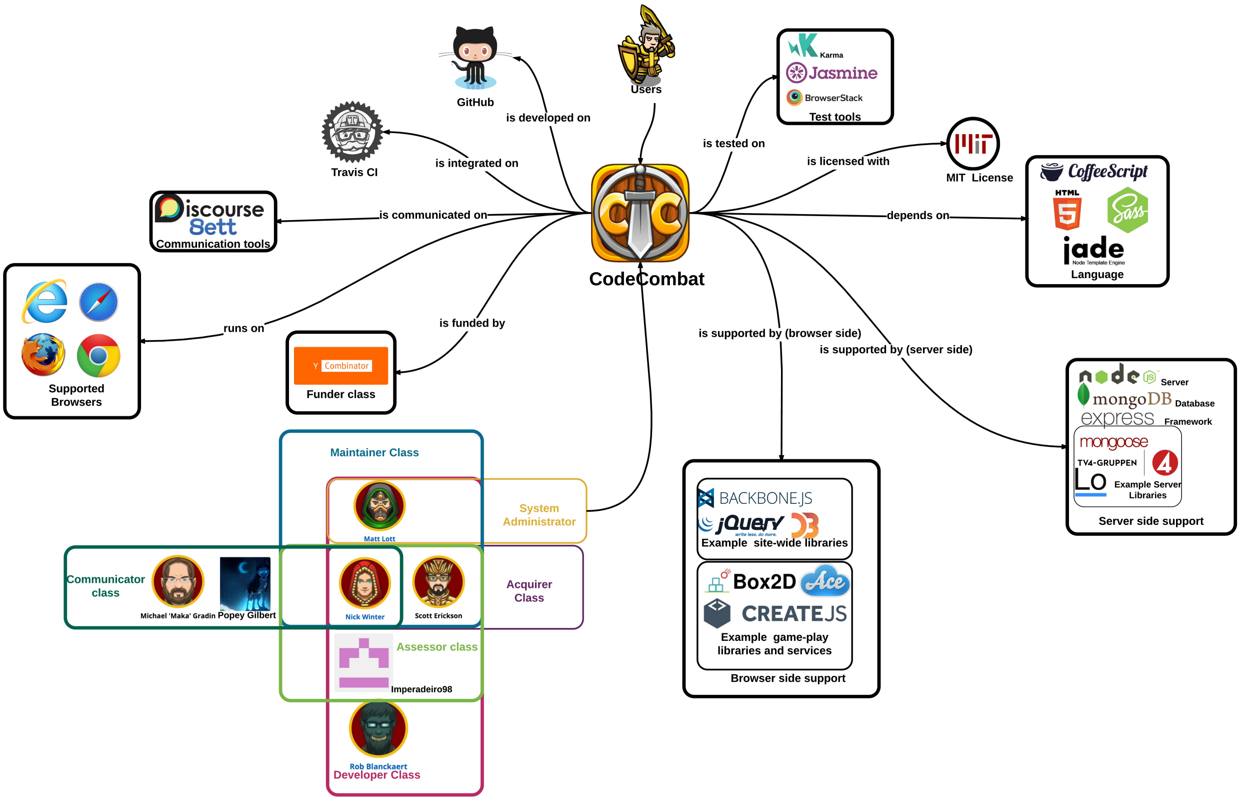


Figure 6 The context view diagram

6. Development View - *The building blocks of the empire.*

The context view showed how the environment and the interaction with the environment looked like. But how does "inside" of the game itself looks like? In other words, how is the game built, which modules are used and how are these modules developed? A development view (based on Rozanki & Woods (2012)) can give answers to these questions.

6.1 Modular structure

CodeCombat is a single-page web application and consists of a client side and a server side. The client side follows the Model-View-Controller (MVC) pattern using Backbone.js. The choice of MVC results in an arhcitecture where new features can be added easily. The server side follows the framework provided by Express.js. In figure 7, a modular structure model is displayed.

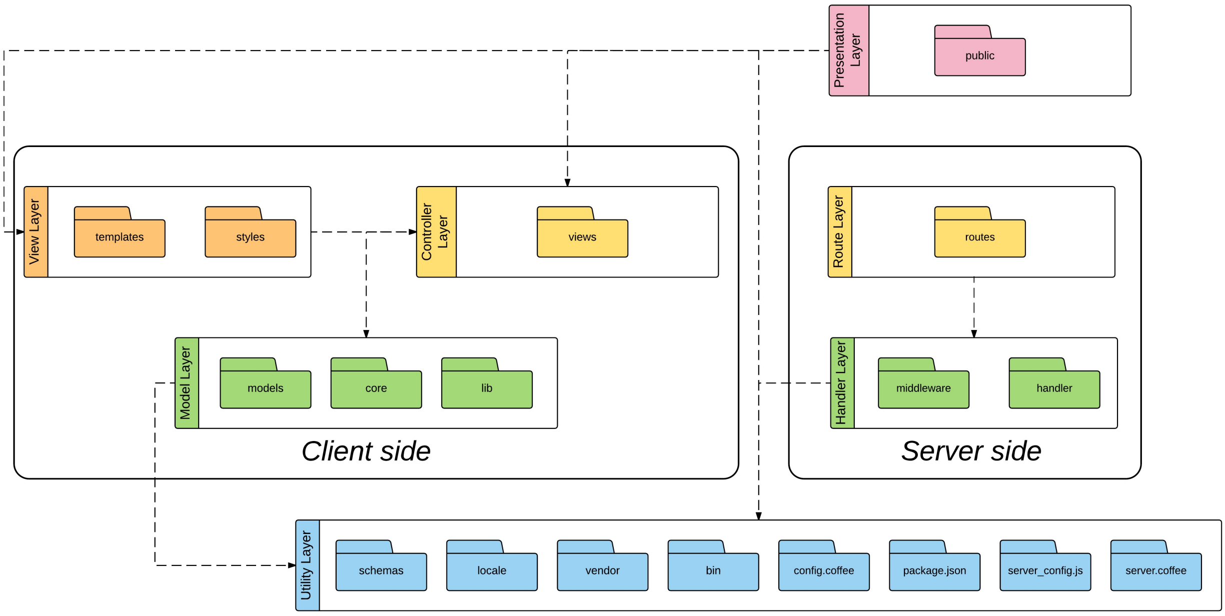


Figure 7 Modular structure of CodeCombat

The **presentation layer** provides users with the final view of the CodeCombat web application via a graphical user interface. It is dependent on the lower layers in the model.

The client side consists of three layers and follows the structure of MVC pattern. The **model layer** contains model components of the CodeCombat client side. The **controller layer** interacts with models and renders the view. The **view layer** is used for producing the web pages for the presentation layer.

The server side consists of two layers. The **route layer** determines how CodeCombat responds to client requests to particular endpoints. The **handler layer** consists of two modules. The middleware module is used for pre-processing of the requests from the clients and the handler module is responsible for processing the requests when certain routes are matched.

The **utility layer** provides utilities and configuration files that can be used in other layers. An example of such a configuration module is the config coffee module.

Dependencies between layers are denoted by arrows in figure 7. The presentation layer has dependencies on the view layer, the controller layer, and the utility layer. The view layer produces the web pages for the presentation layer. The controller layer is responsible for rendering the view. The route layer has a dependency on the handler layer because the route layer determines how CodeCombat responds to client requests to particular endpoints, while handler layer provides functions when the route is matched. The utility layer provides support for other layers.

6.2 Common design models - *The laws in the empire*

Now that it is clear how the modular structure looks like, it is important to know what guidelines maintain this structure. Many developers are working on the game, so some common design models are necessary to keep the structure of the system consistent.

Logging, Internationalization and Database interaction

The developers make use of the following levels of logging messages: `console.log` , `console.info` , `console.warn` and `console.error` . Most of the logging is done with the standards library of Javascript but at the server level, Winston is used.

Internalization is to implement the game in a certain way so it can easily be adapted to specific natural languages. CodeCombat does not want to be dependent of natural languages, and therefore uses the i18next tool to deal with translations.

CodeCombat combines many data from the server together to form the game. Because users are allowed to make their own characters and levels, there is a risk that there is a data overload. Therefore, they use GridFS for storing this data when interacting with the database. GridFS stores data not into single documents, but divides it into parts or so-called chucks. Therefore, they can interact with their data with `Pathmetadata` , instead of interacting with a whole document.

Standardization of Design

The project consists mainly of CoffeeScript files, and contributors are therefore recommended to follow the CoffeeScript Style Guide. The guide presents a collection of best-practices and coding conventions for the CoffeeScript programming language. They also follow the CSS Style Guide from Primer. Additional to these style guides, there are some other guidelines developed by the founders of the CodeCombat team on the CodeCombat wiki. For the use of third-party tools it is recommended to have a look at the currently used third-party tools and use one of these instead of adding another a new one.

Standardization of Testing

Testing is something that can be really helpful in open source projects to establish a robust source code. Unfortunately, CodeCombat has no guidelines on testing. But, CodeCombat does some testing without general guidelines. The CodeCombat Github account is hooked up to TravisCI which runs all the server and client side tests for each commit and pull request. Also, there are some third-party tools used for testing. The tools used include Request and Jasmine. Request is used to test query to the test server. Jasmine is a behavior-driven development framework for testing CoffeeScript.

Overall, the guidelines for both testing and design appear simple and are quite abstract. For a system where some many different people are working on it is important to have some standards for a consistent system design.

6.3 Codeline organization

Figure 8 shows an overview of the directory structure of the CodeCombat directory on Github. It shows the main folder structure as well as the connections between those folders. The app and server folders are the core of the CodeCombat application, where the server folder runs the code on the server.

- The server folder contains many subfolders, and only some of them are shown in the figure.
- The folder app contains the client application source and runs in the browser of the user.
- The folder test contains all the files used for testing.
- Third party resources are located in the vendor folder.
- The bin folder contains the development bash utility scripts and the folder of scripts contain information of the status of the system.
- Lastly, the spec folder contains mostly files for testing and support.

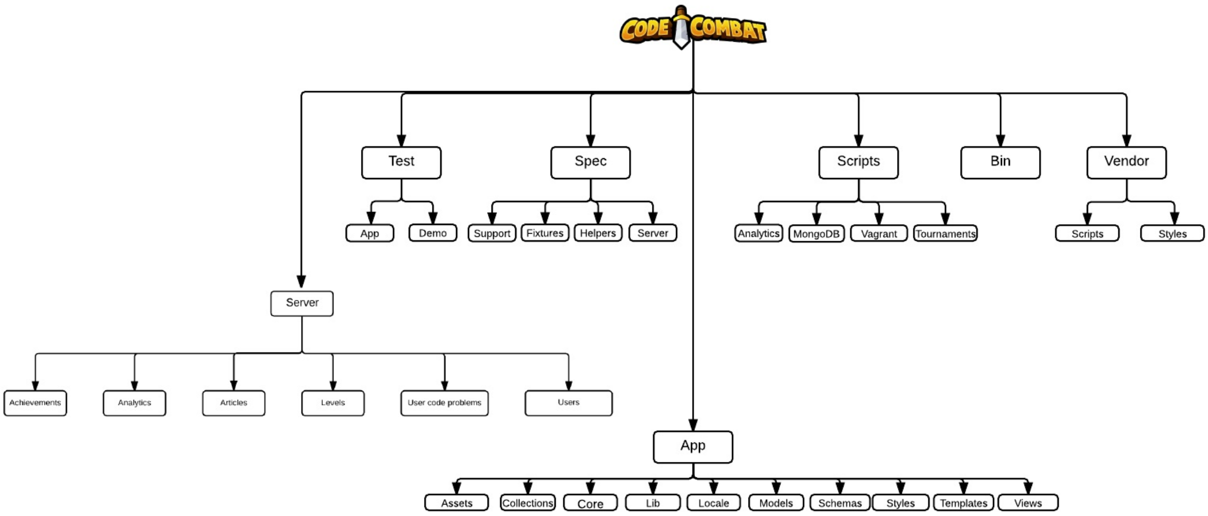


Figure 8 Codeline Organization model

Now that some more technical guidelines have been analyzed in this section, it is also important to focus on some more high-level guidelines which focus on the user interface. since many different types of users interact with the game.

It is therefore important that the system is user-friendly, especially since younger students are working with the game. In the usability perspective this aspect of the system will be analyzed.

7. Usability perspective - *Interaction with the user*

The usability perspective states that the desired quality of a system is that it allows for an effective interaction between the user and the system. The major aspects that are covered include the usability and interaction of the user interface.

7.1 Users

The type of users (as described in earlier sections) are students, teachers and programmers. Each user has a different level of experience. Most users use the system as a final product, but there are some users who make contributions to the project. These contributions are for example designing levels and objects via the editors, or contributing to the source code on GitHub. Those type of users are more experienced with coding/computers since this might be a bit more complicated than just playing the game.

7.2 Touch points

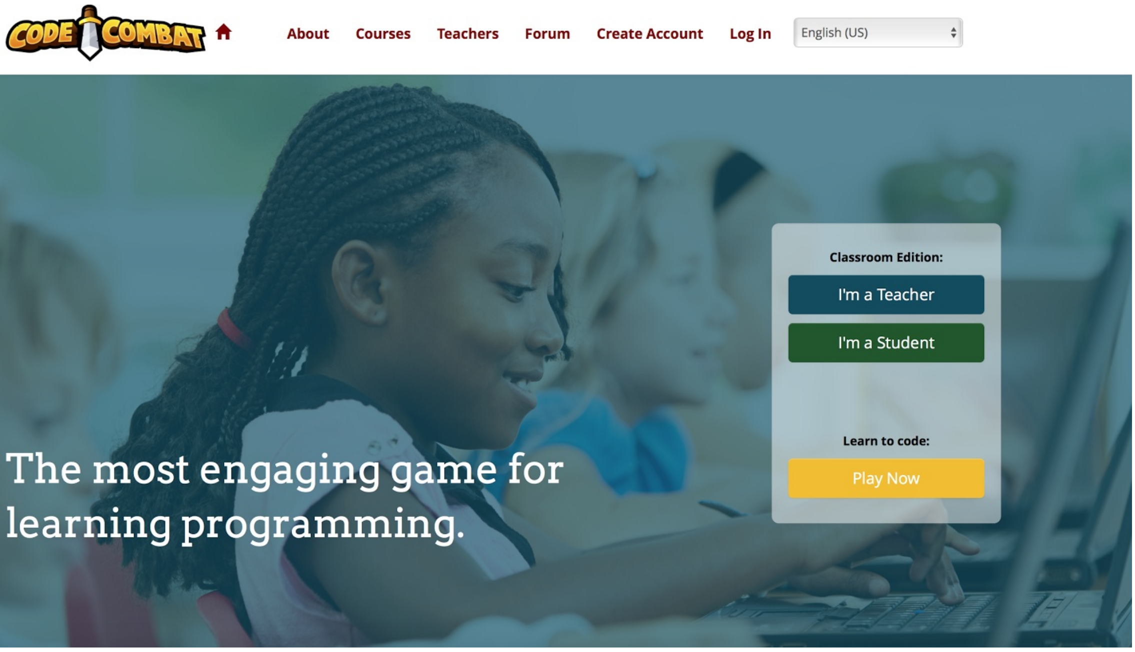
So, how should the user interface manage all these different types of user requirements? Firstly, because of diverse level of user experience, the systems' user interfaces should be simple to use and easy to navigate through. Secondly, it has to take into account the different type of users (e.g. non-students should not have to interact with the student section page).
Thirdly, since the game is used all over the world the game should be translated into many different languages.

To make it easier for the user to navigate to the right page the general systems user interface can be divided into three parts; the homepage of CodeCombat, the game and the editors page. This results in the following setup:

- Homepage
    - Account
    - Info
- Game
- Menus
    - Editor

  - Page layout
- Editors

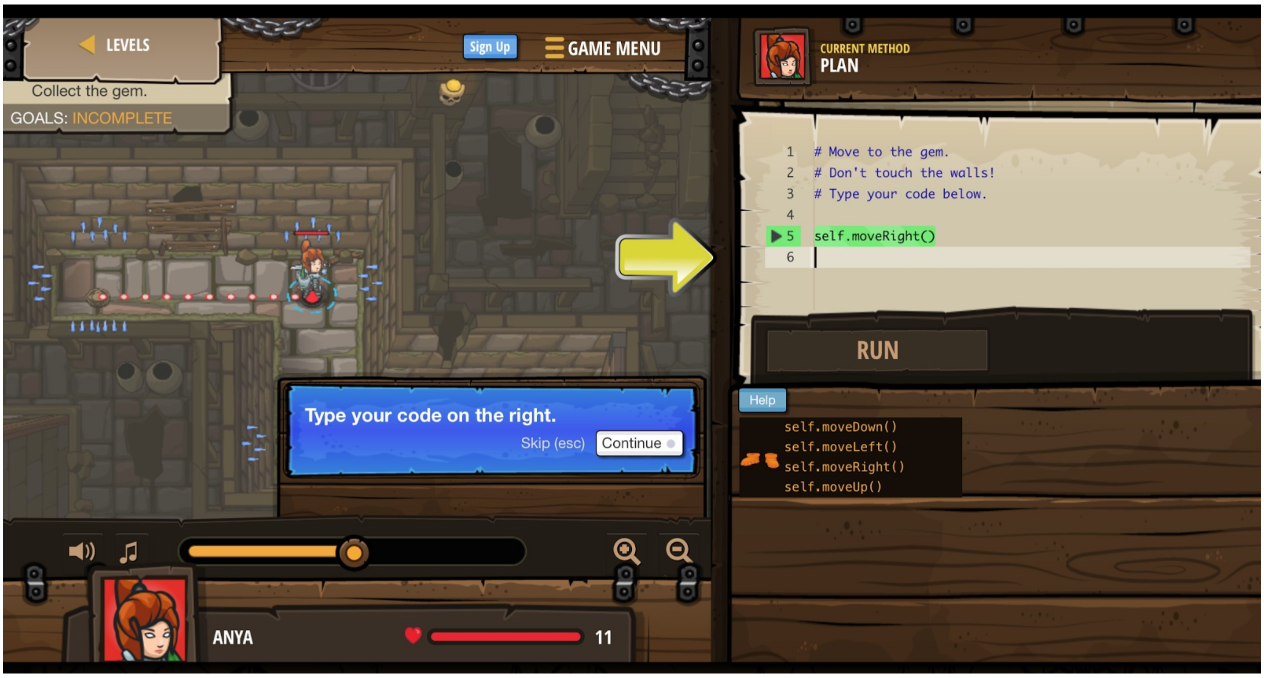
The first user interface is the homepage (see figure 9). On this page, the users can easily navigate to their desired pages, namely the teacher page, the student page , and the regular player page. On all of these pages only very little options are available for the users. This results in an easy navigation for all of the users, whether they are experienced or not.



The most engaging game for learning programming.

Figure 9 The homepage interface

The second user interface (see figure 10) the user can interact with is the in-game interface. This interface helps the user by pointing out where to navigate by showing a pointer to the next level. The most buttons in the in-game interface are straightforward, but some of them are not explained beforehand. This results in that the users have to figure out the meaning of the buttons themselves. Nonetheless, in the first level of the game a tutorial is given on how to use the system.



Collect the gem.

GOALS: INCOMPLETE

Type your code on the right.

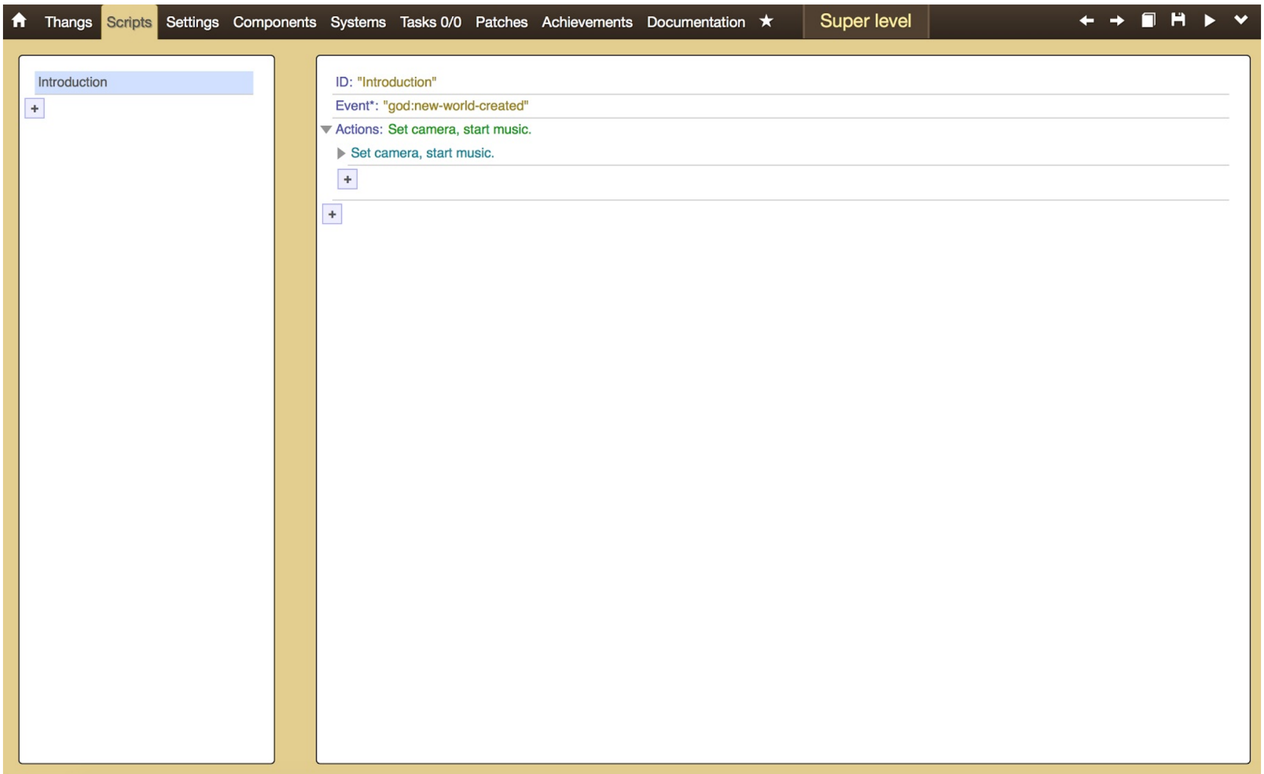
Skip (esc) Continue

```
# Move to the gem.
# Don't touch the walls!
# Type your code below.

self.moveRight()

```

```
self.moveDown()
self.moveLeft()
self.moveRight()
self.moveUp()
```

Figure 10 The in-game interface with the tutorial

The third user interface (see figure 11) the user can interact with is the editors page. The editors page user interface is not easy to understand for most users and therefore requires some additional reading on the wiki, or simply some trial and error. Furthermore, it is difficult to locate to editors page itself, since to get there the user has to navigate to the bottom of the homepage and find the small ‘community’ link. This link directs the user to the editor page. The name of this link is not straightforward and easy to understand.



ID: "Introduction"

Event*: "god:new-world-created"

Actions: Set camera, start music.

Set camera, start music.

Figure 11 The editor page user interface

The overall concept of CodeCombat is about teaching users how to code. The interface of the system takes into account the variety of users by the ease of use and the thorough explanation at the first time using the game. Generally, the interfaces can be considered as self-explanatory, but some interfaces are less straightforward, like the editor page.

8. Variability Perspective - *Variability of the features*

8.1 Feature model

Some of the features described in section 2 of this chapter have a high variability. The features and their variability are displayed in figure 12.

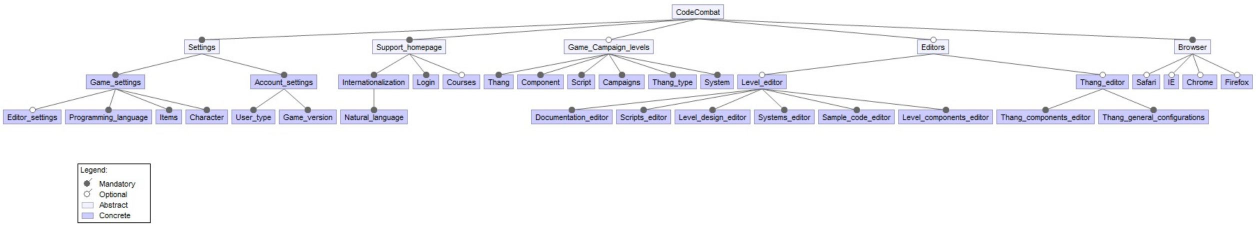


Figure 12 Feature model.

The model displays the *settings* feature variability on the left side, which consists of *game settings* and *account settings*. *Game settings* mainly includes the chosen *game character* and *items*, the *settings* of the editor (the part of the UI where the user enters code to control the game character), and the chosen *programming language*. Account settings include *user type* and *game version*. The game has special environments for the different user types (teacher or student) and offers a free and paid version.

The support homepage variability consists of the *internationalization* feature (there are over 40 different natural languages available), *login* options (Google, Facebook, or CodeCombat itself) and *courses*. The feature *courses* is meant for both the teachers and students and consists of different *classes*_ where various programming skills can be learned and taught.

The *game campaign levels* feature variation depends on the chosen level. As described earlier, CodeCombat works with the Thang Component System (see section 2).
Each level consists of different *thangs* , *components* , *scripts* and *systems*. All of these items together form a level in which the user can play.

All these aspects can be modified via the feature *editors*, as described in section 2.

8.2 Features binding

How is variability configurated? Different implementation techniques to determine the different binding times of features are discussed in this section.

During compile-time binding

In CodeCombat features are implemented independently without class refinement, but rather with class extensions. Almost all models in the folder `codecombat/app/models/` extend from the CocoModel. For example, `Level.coffee` in the folder `codecombat/app/models/` focuses on the feature *GameCampaignLevels* and extends from the class `codecombat/app/models/CocoModel.coffee`, which is also a subclasses from Backbone.Model. Therefore, codecombat does not hold the two compile-time variabilities, preprocessor, and feature-oriented programming.

Furthermore, CodeCombat is mainly written in CoffeeScript, which means it requires precompilation to javascript. The precompilation can be involved with some compile-time feature binding, in which decisions are made about which files to include to generate javascript.

During load-time

Load-time binding involved with command-line parameters and configuration files. Variations are available after compilation and before deployment. An example of the load-time variability of feature internationalization is implemented in server_setup.coffee file, which can be found in the repository in CodeCombat. The code below (figure 13) realizes the feature Internationalization according to the country code of domain name. For example, the language of codecombat china with country code *cn* is set to be Chinese.

```
setupCountryRedirectMiddleware = (app, country="china", countryCode="CN", languageCode="zh", serverID="tokyo") ->
  shouldRedirectToCountryServer = (req) ->
    speaksLanguage = _.any req.acceptedLanguages, (language) -> language.indexOf languageCode isnt -1
    unless config[serverID]
      ip = req.headers['x-forwarded-for'] or req.connection.remoteAddress
      ip = ip?.split(/,? /)[0]  # If there are two IP addresses, say because of CloudFlare, we just take the first.
      geo = geoip.lookup(ip)
      #if speaksLanguage or geo?.country is countryCode
      #  log.info("Should we redirect to #{serverID} server? speaksLanguage: #{speaksLanguage}, acceptedLanguages: #{req.acceptedLanguages}, ip: #{ip}, geo: #{geo} -- so redirecting? #{geo?.country is 'CN' and speaksLanguage}")
      return geo?.country is countryCode and speaksLanguage
    else
      #log.info("We are on #{serverID} server. speaksLanguage: #{speaksLanguage}, acceptedLanguages: #{req.acceptedLanguages[0]}")
      req.country = country if speaksLanguage
      return false  # If the user is already redirected, don't redirect them!

  app.use (req, res, next) ->
    if shouldRedirectToCountryServer req
      res.writeHead 302, "Location": config[country + 'Domain'] + req.url
      res.end()
    else
      next()
```

Figure 13 Example of source code during load time

During run-time

With run-time bindings, decisions can be changed during execution time. Since the features implemented at run-time binding follows the same pattern set by MVC, in this section only several features have been analyzed in detail.

The feature Login enables users to sign in their user account with their CodeCombat account, Facebook account, or Google account. When users try to log in codecombat from browser side, the server will first load the three account selections for users. After the selection, the request is sent from the browser to the server to enable access token of different login methods for users.

The feature *Game Characters* enables users to choose a game character from various game characters. With requests from the users and responses from the server, users can change game characters during running time.

Now that the whole system has been analyzed, it is interesting to look at how the system is put into practice. This is done by looking at the technical debt. Technical debt is a metaphor used to describe legacy problems that need to be solved or obsolete components that needs to be updated.

9. Technical Debt - *How well is the system implemented?*

The technical debt analysis focuses on aging libraries, documentation, defects, and refactoring. The identified technical debts are as follows:

Aging Libraries

Using aged libraries, especially no longer maintained ones, is exposing the system to risks and the vulnerability of the system will naturally increase. In table 2 an overview of all of the aging libraries is shown.

Libraries	Version Used	Latest Version
Backbone.js	1.1.0	1.2.3
jQuery	2.1.4	2.2.1
moment.js	2.5.1	2.11.2
i18next	1.7.3	v2
d3	3.4.13	3.5.16
Modernizr	2.8.3	3.3.1
Firebase	1.0.24	2.4.1

Table 2 Aging libraries

It can be concluded that some important libraries are outdated. Using software like Bundler, improves the overall insight in the status of the used libraries for developers. Therefore, CodeCombat is strongly advised to use such a service.

Documentation

Documentation is a great assistant to a number of stakeholders (e.g. users, developers, and maintainers). However, the documentation of CodeCombat is not centralized in one place and far from sufficient, because most of the documentation available describes the system very briefly. This brings difficulty to playing, developing, and maintaining the game.

Defects

The most easy technical debt to find is the defects in the code, which are listed on GitHub issue tab. Some bugs are reported more than two years ago but still unfixed, like issue #28 and issue #311. They indicate the presence of technical debt and need to be fixed as soon as possible.

Refactoring

By looking at some pull request, one could conclude that CodeCombat refactors their code. In pull request #1394 and pull request #1379 rubenvereecken did some refactoring. This is only an example of one refactoring of code, and since there is not much discussion on the pull request, it is hard to draw a conclusion of their refactoring policy.

10. Conclusion and recommendations

The architectural analysis results in some interesting findings. CodeCombat is a game with many different features and functionalities, which teaches users how to code by playing the regular game and also enable users to create their own content via the editors. The functional view shows that the core functionality of CodeCombat is the regular game play, which depends on various functional elements with different responsibilities. The context view shows the outside environment of the system without which CodeCombat cannot be successfully built, tested, and operated.

Many different types of stakeholders are involved in the development of the system. The most important stakeholders are the founders, Nick Winter and Scott Erickson. They are still involved in the development of the game by merging most of the pull requests. There are also many contributors actively contributing to the system on the Github repository. These contributors all have different ways of working, which therefore require some guidelines to realize a consistent design of the system. The development view shows that although CodeCombat makes use of some design and testing guidelines, the level of detail of these guidelines could be improved. However, the development view presented also a good aspect of the system; the system's’ modular structure with a MVC architecture is well organized.

The usability perspective showed that most of the interfaces are user-friendly, but the editor interface lacks some user-friendliness since the interface is sometimes messy and unclear. The variability perspective shows that features presented to the users are mainly configured in the run-time. Furthermore, technical debt is another aspect that needs the attention of developers. Some examples of technical debt are aging libraries and inadequate documentation. These problems need to be solved since it would potentially bring obstacles to the development of the system.

All in all, CodeCombat is a structured project which has attracted a large amount of users. Developers who want to contribute to the project could focus on standardization, technical debt and the usability of some of the interfaces.

References

[1] Nick Rozanski and Eoin Woods. 2012. Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives. Addison-Wesley Professional.

[2] Apel, Sven, Don S. Batory, Christian Kästner, and Gunter Saake. Feature-oriented Software Product Lines: Concepts and Implementation. Retrieved on 5-3-2016 from http://link.springer.com/book/10.1007/978-3-642-37521-7

[3] CodeCombat Homepage. Retrieved on 18-2-2016 from https://codecombat.com/

[4] CodeCombat GitHub Repository. Retrieved on 5-3-2016 from https://github.com/codecombat/codecombat

[5] CodeCombat Blog. Retrieved on 18-02-2016 from http://blog.codecombat.com/

[6] CodeCombat Facebook Page. Retrieved on 5-3-2016 from https://www.facebook.com/codecombat

[7] CodeCombat Wiki Home. Retrieved on 18-2-2016 from https://github.com/codecombat/codecombat/wiki

[8] Why Basic Coding Should Be a Mandatory Class in Junior High. Retrieved on 30-3-2016 from http://time.com/2881453/programming-in-schools/

[9] Managing technical debt. Retrieved on 5-3-2016 from: https://18f.gsa.gov/2015/10/05/managing-technical-debt/

Appendix

Contributing to CodeCombat

Besides learning how to code by using CodeCombat, users can also learn how to code by contributing to it. This way it is possible to code coding while working on a real project and this makes CodeCombat even better.

How to start contributing

Making contributions can be done in several ways, but is mostly done by forking the Github repository and work on issues. Another way is by using one of the editors that are integrated on the CodeCombat web page. For more information about the editors it's best to look at the wiki page of CodeCombat.

Before any contribution can be made the user has to sign the Contributor License Agreement. After this the user can set up the CodeCombat environment and start making contributions.

For working on translations for CodeCombat users can also go to the Diplomat page and go to the translation page for translating the levels or select the language they want in the list of languages to translate the interface and website. Selecting a language from the list brings them to the respective file for that language on GitHub, where they can edit it online and submit a pull request directly.

D3.js - Data-Driven Documents

Authors: M. Dunnewind, P. Gupta, S. van Schooten and P. van der Veeken

Abstract

D3.js is a JavaScript library which enables developers to manipulate a website's HTML, DOM and CSS styling. The core strengths of the library are the specialized manipulation functionalities which allow for binding DOM elements to a dataset. This data-binding, along with a multitude of additional functionality, makes it easy to create beautiful, interactive visualizations. The library is a long standing and proven project which is used by developers, data scientists and journalists alike to create a diverse range of different visualizations about all kinds of topics. In order to stay future-proof, relevant and to keep-up with new developments, constant work is done on the architecture. A prime example of this continued development is the upcoming release of version 4.0. This release features a rewrite of the entire library structure, to make it completely decoupled and suitable for extensive customization.

Table of Contents

Introduction

D3.js (Data-Driven Documents) is a long standing and widely used visualization library written in Javascript. It enables developers to easily manipulate a webpage's HTML DOM (Document Object Model) and CSS styling to create stunning visualizations. The central concept on which the library is built, is that it should be possible to easily bind data to DOM elements. This data-binding, along with a multitude of additional functionality, makes it easy to create beautiful, interactive visualizations.

The library has existed for several years now, and during its lifetime it has went through several major version changes. These changes range from architectural modifications to natural evolution in order to keep up with the changing needs of the library's users. This becomes most obvious when looking at the upcoming major version change, in which the structure of the library is completely overhauled to make it more modular and customizable.

In this chapter we will try to demonstrate how powerful and well-suited D3.js is for visualizing data. Additionally, we will detail how the library has evolved throughout the years, how the underlying architecture of the library functions and who the different people and organizations are who have a vested interest in this library.

D3.js

Before we dive deeper into the architectural intricacies of this visualization library, you as a reader should first gain a better understanding of how D3.js is used by hundreds of thousands of developers around the world to create breathtaking visualizations. This section will attempt to give an insight into the ways D3.js facilitates creating these visualizations and what the general workflow is for doing so.

Creating Visualizations with D3.js

D3.js works by allowing its users to easily attach data to DOM elements. By harnessing the power of CSS, SVG and HTML, it becomes possible to create stunning, interactive visualizations.

Ben Fry has described the process of creating visualizations from data as a series of subsequent steps [9]:

1. Acquire
2. Parse
3. Filter
4. Mine
5. Represent
6. Refine
7. Interact

Step 1 will not be very relevant to using D3.js, because acquiring an actual dataset of which to create a visualization is something the user needs to themselves, obviously.

Parse

Datasets are commonly formatted as either comma-separated values or tab-separated values. For both these formats D3.js provides functionality to parse them.

A concrete example: imagine that the grades of participants of the software architecture course are stored in a file called grades.csv which is structured in the following manner:

```
group_name, language, members, grade
Team-d3, Javascript, 4, 9
....
```

We now want to visualize if the choice of language is correlated with a group's grade. The first step is then to parse the grades.csv file. Which is done like so:

```
var grades = []
d3.csv("grades.csv", function(data)
{
    /*data is now an array of json objects containing the data
    from the csv*/
    data.forEach(function(d))
    {
        /*each d is one line of the csv file represented as a
        json object*/
        grades.push({
        "group":d.group_name,
        "language":d.language,
        "members":d.members,
        "grade": d.grade});
    })
});
```

Notice in the code example above, how seamless the transition is from parsing a dataset to mapping the resulting data.

Filter

When a dataset is parsed, it is rarely the case that it contains only the data one is interested in. In such cases, the data needs to be filtered in order to trim away the irrelevant entries. D3.js provides its users with the powerful `filter()` function for doing exactly that.

Continuing with the example above, say we are only interested in the languages Javascript, C++ and Ruby, then we need to filter out all the other languages. Furthermore, for the visualization we're creating, the group's name and number of members are not very relevant, therefore we need to filter those out as well.

```
grades = grades.filter(function(d)
{
    //Filter for only our desired languages
    var lang = d.language;
    if(lang == "Javascript" || lang == "C++" || lang == "ruby")
    {
        //Return only the fields we're interested in
        return {"language":d.language, "grade":d.grade};
    }
});
`
```

We now have an array of filtered data containing only the variables we're going to use for a visualization.

Mine

The nest feature of D3.js allows for elements in an array to be grouped in a tree structure much like the GROUP BY operator in SQL. Unlike the SQL operator though, Nests allow for multiple levels of grouping. Mining data is all about extracting useful information from a dataset. Nests make it effortless to categorize data in any way the user wants, and for this reason D3.js is a powerful tool to use for the mining step in creating a visualization.

An example: The data obtained from the filter example above, now looks something like this:

```
var grades = [
    {language:"Javascript", grade:9,
    {language:"C++", grade:8},
    {language:"Ruby", grade:5},
    {language:"Javascript", grade:7},
    {language:"C++", grade:6},
];
```

We now want to group the data by programming language and then order by grade, to see if the choice of programming language has an influence on the grade of a team. This is done easily with the following code:

```
var nest = d3.nest()
    .key(function(d) { return d.language;})
    .key(function(d) { return d.grade;}).sortKeys(d3.descending)
    .entries(grades);
```

which produces formatted data, like so:

```
var nest = [
    {key: 'Javascript', values: [
            {key: '9', values:
                [{language: 'Javascript', grade: 9}]
            },
            {key: '7', values: [
                {language: 'Javascript', grade: 7}]
            }]
    },
    {key: 'C++', values: [
            {key: '8', values:
                [{language: 'C++', grade: 8}]
            },
            {key: '6', values:
                [{language: 'C++', grade: 6}]
            }]
    },
    {key: 'Ruby', values: [
            {key: '5', values:
                [{language: 'Ruby', grade: 5}]
            }]
    }];
```

This example might be a bit trivial, but it should be clear that grouping data in this way is exceptionally useful when working with large datasets with a lot of variables.

Represent

In this step of the visualization process, one is supposed to choose in what way they want to represent the data. Common representations are graphs, lists, trees, etc. Whatever choice the user decides to make, d3.js will provide a framework which will make the most common obstacles (i.e creating scales, interpolating points) of each representation less of a chore.

The author of D3 maintains a gallery which showcases a great diversity of the types of visualizations [17]. An important take-away from this gallery is that visualizing data is most-definitely not restricted to just charts and graphs, but can instead be done in so many different ways. With the most important rule being: Explain and convey information in the simplest way possible, but not simpler.

Refine

Refinement entails making the chosen representation as comprehensive and unambiguous as possible. D3.js doesn't necessarily make this step easier because refinement is mostly a subjective step, however it does make applying colour scales, formatting timestamps or easing lines, for example, trivial.

Interact

D3.js provides a whole host of functionality to create transitions and animations easily. It is therefore at this step that D3.js is arguably at its most effective. An outstanding example of how interactivity can make a visualization more effective, is the scatter matrix as can be seen in image 1.

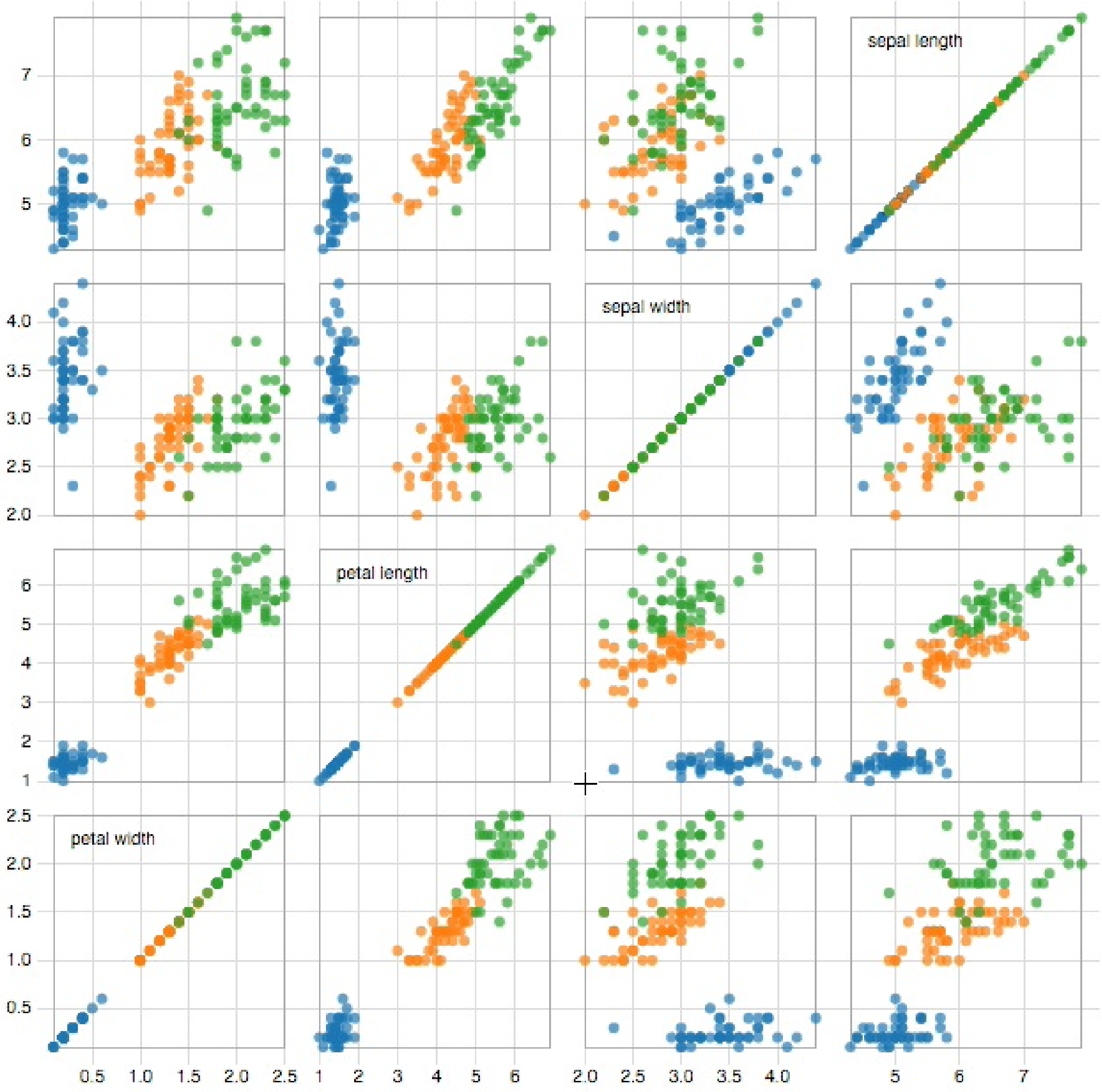


Image 1: click image to go to interactive visualization

Peripheral technologies

The D3 library doesn't exist in a vacuum, it relies on some fundamental technologies and standards without which it couldn't exist. These technologies and standards are:

HTML(5)

As mentioned above, D3.js works by allowing users to attach data to DOM elements. The DOM is an integral part of the HTML standard and as such, D3.js is inextricably entwined with it.

CSS(3)

D3.js makes use of CSS-style selectors to pick elements on which to operate. Furthermore, the idiomatic way of styling elements in a visualization created with D3, is by applying CSS rules to them. For these reasons, D3.js couldn't exist without the CSS core technology.

SVG

D3 is at its best when rendering visuals as Scalable Vector Graphics. SVG is a text-based image format. Meaning, you can specify what an SVG image should look like by writing simple markup code, not unlike HTML tags. SVG code can even be included directly within any HTML document.

For example, drawing a circle can be done by embedding SVG code directly into HTML, like so:

```
<svg height="50" width="50">
    <circle cx="25" cy="25" r="25" fill="#FF3D00" />
</svg>
```

Which results in an image as displayed in image 2.

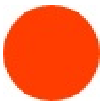


Image 2

Users of D3 are not required to use SVG with D3, but nearly everyone agrees that SVG provides a range of visual opportunities that aren’t possible with regular HTML elements.

Performance

Since D3.js is built upon a series of injectors (see the architecture section), the length and complexity of the injection chain is one of the main variables in terms of performance. To define how much a visualisation costs performance-wise, certain aspects have to be measured for that specific case, such as response time (time to process a single element), throughput (time after another element can be processed, or the 'wait' time) and latency (the time between a change in the dataset and the first form of output). Based on these metrics more optimized solutions can be created which might even take advantage of internal patterns for acceleration.

There are four main components that play a part in the processing chain of D3.js:

- **Input data**: collection of entries that contain the information that is needed to create the visualisation.
- **Element binding**: the informal relation between a data entry and an element in the DOM.
- **Method chain**: the method chain that extracts element parameters based on the data.
- **DOM** (Document Object Model): the document elements that the entries are bound to through the element binding.

These processes are easier to understand when visualized as in image 3.

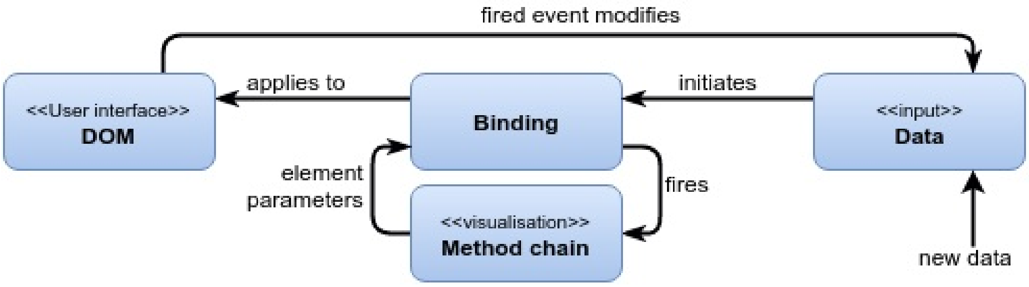


Image 3

As can be seen in the diagram, the method chain is invoked on every action, where entries are either edited, created or removed. This means that this is the first location in which to look for possible speed-ups. The method chain is unique for every visualisation, so the metrics defined above (response time, throughput and latency) can not be compared with other visualisations without taking the difference in the chain into account. For simpler examples, the response time can be fairly linear on the size of the input, meaning that this metric is (at least partially) uncorrelated with the input size. Though sometimes the complete dataset has to be analysed at some (or multiple) point(s) in the chain, this will cause a more exponential behaviour. The same goes for the throughput, though memory and data organisation might be more important here. The latency is affected by almost all elements in the processing chain, since it is measured from input event to first binding modification/instantiation, though the overlap is not complete.

All in all, this means the performance for a given visualisation mostly depends on the complexity and length of the method chain, input size and hardware available to the browser for rendering. This is mainly due to the fact that most of the injectable methods in the chain have a fairly linear runtime. Combining certain non-linear ones will result in a non-linear performance as expected. However, when dealing with very large amounts of data, the browser rendering performance will start being a bigger and bigger issue, since that has actual physical limitations in contrast to the runtime limitations of the method chain.

Stakeholders

The owner and maintainer of the D3.js library is Michael Bostock [7], whose vision it was to utilize DOM manipulation for creating visualizations. His work on D3.js was for a large part influenced by his previous work on Protovis (based on JQuery) and his advisor's prior visualisation toolkits, Prefuse and Flare. The inspiration behind D3.js was to extend the limits of visualization and empower developers with full access and control to the DOM, thereby allowing them to communicate their gathered-data and analysis efficiently. Bostock [8] argues that "visualisation is more than a tool for finding patterns in data." And so D3.js was developed to serve as an enabler for effective, interactive and meaningful visualisation of statistical and analytical data, rather just than an eye-catching graphical representation.

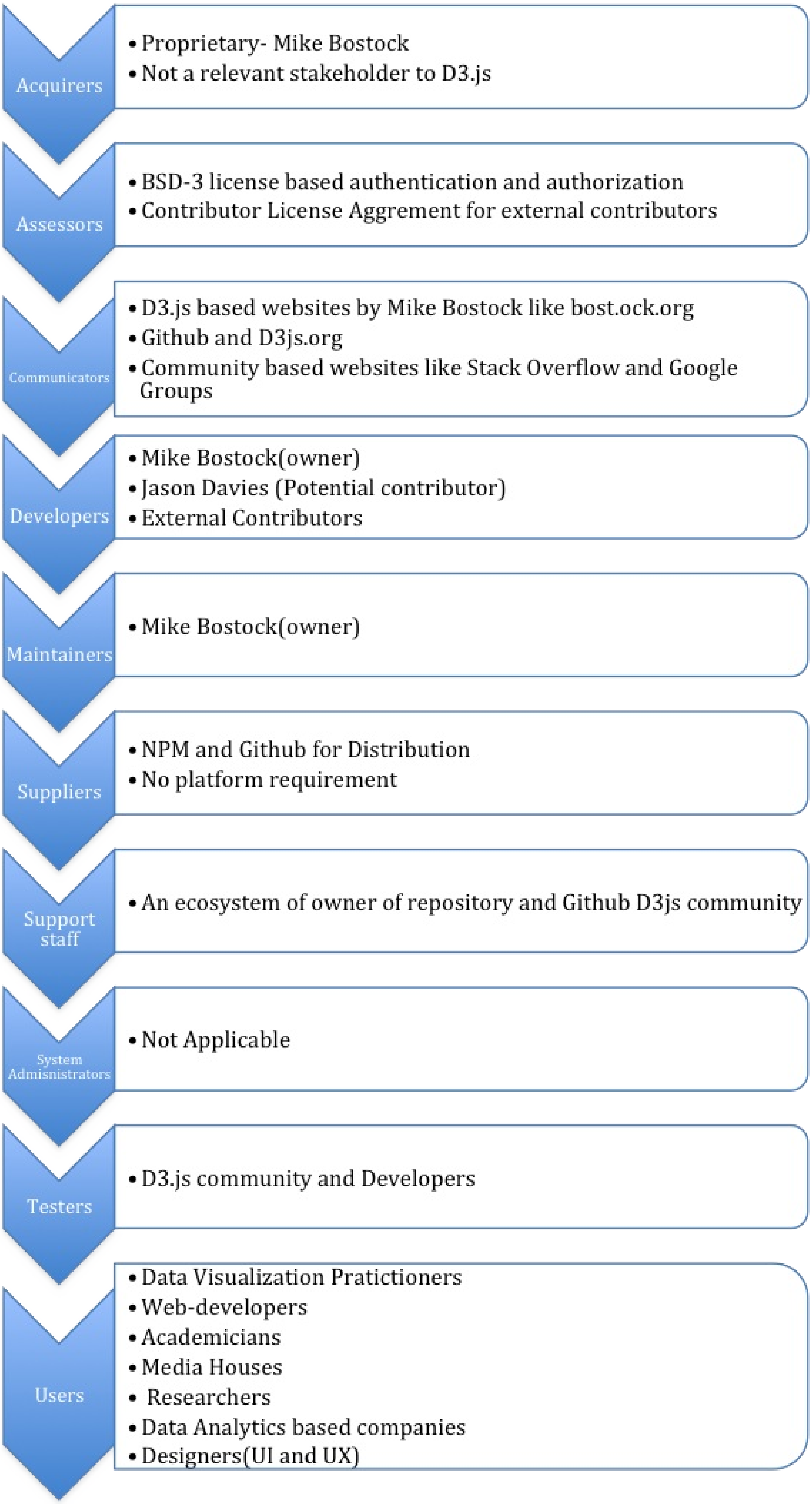
A majority of the contributions to D3.js come from Michael Bostock and Jason Davies, furthermore an active network comprising of D3 communities on Github, Stack-Overflow and Google Groups have made D3 much more accessible and welcoming to the external contributors. In the words of Michael Bostock[2], "I think D3 needs a better story around extensibility and external contributors. This is part of the motivation for breaking it up into smaller modules for 4.0." From this statement it becomes evident that Michael is considerate towards the immense support and help pouring in (also in the form of tutorials, examples, books etc) from D3 practitioners. D3 4.0 is thus aimed to increase reuse and external contributions, which is a must for making an open-source platform sustainable.

D3 enjoys widespread popularity with an average of 9000 downloads [4] per week. This prominence is a cumulative output of extensive documentation and examples, great community support and the approachability of Michael Bostock himself. The target audience is an ever-increasing number of web-developers, data-visualisation practitioners, small companies and start-ups aiming for a big leap in their knowledge and growth. In conjunction with these, there are various market giants [2]. like Microsoft (Microsoft BI makes use of D3) which have employed D3 for interactive data visualisation. What makes D3 an attractive choice for numerous companies is its graphical flexibility in data representation and its ability to take up various roles. Several tools, such as exploratory tools, data mining tools, communication tools and analytics tools can be designed with it as basis. Mentioned below[3][6] are the companies whom are making extensive use of D3.js in media visualizations, fashion reports and much more

- Open Street map: Editing open street maps
- Dataviz:For communication and business tools
- Chart.io : Providing Business Dashboard solutions
- Plotly : Rendering Plots
- DataMeer : Representing Big data exploration
- New York Times :Rendering Graphs and statistical analysis.

Considering the Large scale visualization capacity[5] of D3.js (which can be utilized in visualizing Bigdata workflow), it is of huge interest for the Big data companies like Datameer, SAP HANA, Palatfora Clear Story, Trifacta etc. An example of BigQuery Big Data Visualization implemented using d3.js can be found here. D3.js can also serve as a real-time visualization tool for ambitious projects like Human Genome Projects and DOPA implementations.

In summary, D3.js stakeholders can be classified into below mentioned categories as per Roznaski and Woods[10] which is shown in image 4.



Acquirers

- Proprietary- Mike Bostock
- Not a relevant stakeholder to D3.js

Assessors

- BSD-3 license based authentication and authorization
- Contributor License Aggrement for external contributors

Communicators

- D3.js based websites by Mike Bostock like bost.ock.org
- Github and D3js.org
- Community based websites like Stack Overflow and Google Groups

Developers

- Mike Bostock(owner)
- Jason Davies (Potential contributor)
- External Contributors

Maintainers

- Mike Bostock(owner)

Suppliers

- NPM and Github for Distribution
- No platform requirement

Support staff

- An ecosystem of owner of repository and Github D3js community

System Admisnistrators

- Not Applicable

Testers

- D3.js community and Developers

Users

- Data Visualization Pratictioners
- Web-developers
- Academicians
- Media Houses
- Researchers
- Data Analytics based companies
- Designers(UI and UX)

Image 4

Competitors

D3.js is undoubtedly one of the best data visualization libraries in the current scenario. As per a notable data visualisation specialist Moritz Stefaner[15],no one is as good as D3.js especially in the case of SVG based data visualization .There are several other worth-mentioning data visualization tools and java script libraries in competition like Tableau, RAW, Leaflet, Chart JS, Timeline JS etc. The only downside of D3.js (which is ironically the strongest attribute as well) would be the complete freedom given to the data visualizer, there is nothing to recommend or suggest from the D3.js end. In the case, when users are looking for specific data visualization services ,then tools like GEPHI, for network visualization and Chart JS for precise control over charts might come handy [16]. Tableau, another popular visualisation toolkit which does matrix based visualizations differs significantly from D3.js which a general purpose visualization framework.. While D3.js relies on the inherent properties of the browser, then in the situations where browser does not offer extensive functionalities, usage of D3.js might not offer a pleasant experience. Also, initially it might seem more interesting to programmers than common users as writing a code to create a graph can be a cumbersome approach for them. Although ,the users who are willing to migrate to D3.js , they are motivated to experience an amazing user interaction and advanced visualization provided by D3.js.

Architecture

Since D3.js is a framework with which you can build your own visualizations, it is mostly comprised of functions and modules that create elements in a visualization or modify them in one way or another. The basis is a core module that offers document-data binding methods, where the other modules plug into to create the desired graphics. By binding the data elements to elements in the document, D3.js can seamlessly create these visualizations with low overhead. The browser handles the rendering in the same manner as a regular HTML/SVG document and is thus very fast.

Inner workings

Since D3.js is a visualization library and not an application, D3.js depends on certain modules and elements. As will be mentioned later in this section, D3.js is in the middle of a big restructuring which completely changes its underlying architecture. From a functional viewpoint[10], we will be focussing on the new structure rather than the old one, because it will be relevant much longer.

From *Software Systems Architecture*, chapter 16: "*Functional capabilities define what the system is required to do—and, explicitly or implicitly, what it is not required to do.*" [10]. Because D3.js does so many different things, each module of D3.js (as of Version 4) will have its own short description in the table below.

Module	Description	Dependencies
d3-array	The module responsible for transforming data into information, since D3.js can only bind data in array form.	None.
d3-axis	Alleviates the need to manually create axis for charts and tables.	d3-scale, d3-selection, d3-transition
d3-collection	Offers more complex data structures.	None.
d3-color	Provides a wide collection of color manipulation and representation methods.	None.
d3-dispatch	Event emitting and listening module for more decoupled code creation.	None.
d3-dsv	Parser module for delimiter separated values, such as CSV and TSV.	None.
d3-ease	Providing easing functions for transitions and animations.	None.
d3-format	Number formatting made easy.	None.
d3-interpolate	Exposes a number of methods for interpolating two values.	d3-color
d3-path	Easy path rendering, similar to path drawing on canvasses.	None.
d3-polygon	Applying geometric operations on two-dimensional polygons.	None.
d3-quadtree	A module dividing data using a quadtree: a two-dimensional recursive, spatial subdivision.	None.
d3-random	Exposes a number of random number generation methods.	None.
d3-request	Conveniency wrapper for XMLHttpRequest.	d3-dsv, d3-dispatch
d3-scale	Encoding for abstract to concrete data mapping.	d3-array, d3-collection, d3-color, d3-format, d3-interpolate, d3-time, d3-time-format

d3-selection	DOM transformation by selecting and joining data.	None.
d3-shape	Graphical primitives, such as lines and areas.	d3-path
d3-time	A calculator that can cope with the human readable time definitions.	None.
d3-time-format	Date parser.	d3-time
d3-timer	A low overhead queueing mechanism for managing thousands of concurrent updates and animations.	None.
d3-transition	Animated transitions for D3.js selections.	d3-ease, d3-timer, d3-interpolate, d3-selection
d3-voronoi	Voronoi computation for a set of two-dimensional points.	None.
d3-brush	Wrapper for easy selection of data points in a graphical manner.	d3-scale, d3-selection
d3-hierarchy	Layouting algorithms for visualizing hierarchical data.	None.
d3-geo	Wrapper for geographical calculations and projections.	None.

Current status (v3.5)

As is mentioned above, the current state is based on a core module and (mandatory) plugins, all wrapped in one large package. Though it is possible to compile you own library with only the plugins needed, this option is not offered or even documented. The current architecture is based on modules and dependencies, which are visualized in image 5.

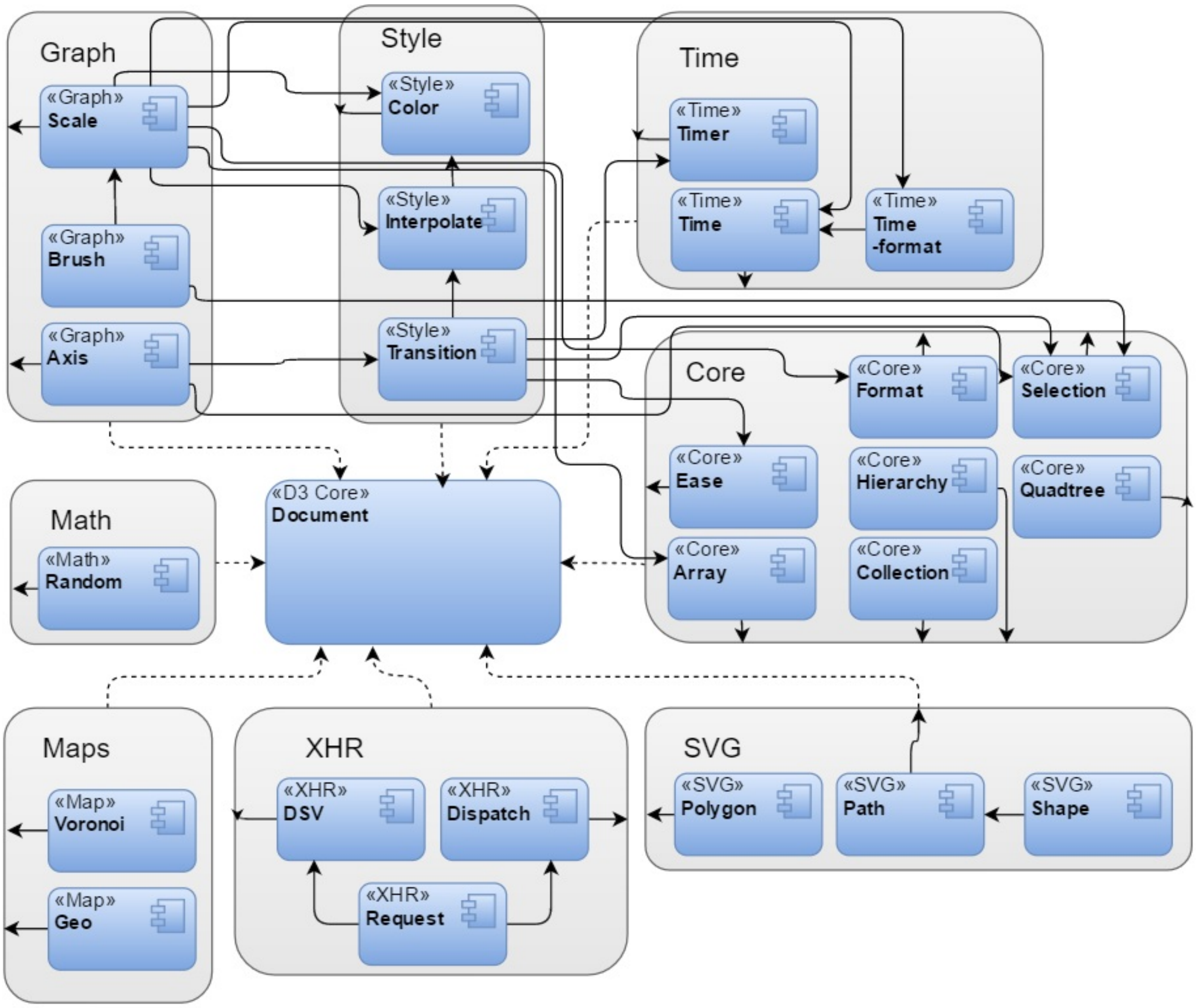


Image 5: modules and dependencies

It is obvious that D3.js uses an architecture that is easily expandable and improvable. Each module in the graph above has a responsibility that can be clearly derived from the module name. The archetypes (or boundaries) are depicted above the name to indicate more implicit relations between the modules. Creating certain kinds visualizations requires certain modules, which have to be used in a manner and order that allows the injection into other modules. This order has to be in accord with the way build-up of a visualisation is intended.

Most of these modules will return functions that can be applied and chained to create whatever desired effect the developer wishes. The modules are built as a system of *injectors*, which is a software design pattern that resolves dependencies in an inverse manner [1], resulting in an easy to comprehend and pluggable library.

Upcoming release (v4.0)

With the upcoming release, the biggest change will be the fact that the library is pluggable. This means less overhead and faster loading times, due to the fact that only the modules which are actually used, are loaded. The plugins load themselves into the already loaded d3 global variable, which makes a module's functionalities available to the end-user. The only requirement for this system to function, is that the d3 core is initialized first, and that then all the plugins (in no particular order) are loaded.

Although this new architecture reduces cost and overhead, it does increase the number of dependencies of a any given project. However, this downside can also be viewed as an invitation for a developer to write their own injectors and use them as d3 plugins. When a certain structure or organisation is used a lot in a project, it is a viable possibility to roll their own plugin which pre-injects d3 with their own definitions, removing the need to re-implement certain functionality.

One of the biggest up-sides of the restructuring will not be for the end-users, but for the developers; testing and building will not fail or result in monolithic, modules and the core can be bundled using something like WebPack. Modules can be used separately as injectors for other systems, so you can use the advanced color scale generation methods for other projects too. Along with the upgrade to EcmaScript 6, d3 has become much friendlier to extend and use.

Evolution

In this section we will detail the evolution of the D3.js library. The evolution cycle is the process which a system undergoes in order to adapt to changes in the software development process. As discussed in Rozanski and Woods, a flexible system should be able to deal with all the possible types of changes that it may experience during its lifetime [10]. Just like the previous section, this section will for one part focus on the current version (v3.5) and for the other part on the upcoming version (v4.0).

Evolution Needs

D3.js is a continuously developed visualization framework based of Michael Bostocks PhD thesis, it was first registered as open source project on Github in late 2010. At that time the Google Closure Compiler [11] was still used to optimize the different code parts and combine them into a single source file. However the popularity of D3.js grew quickly and so did the demand for a compiler which would generate smaller and more performant source files. In March 2011 the team switched to using UglifyJS [12] for their build process. This change was the only noteworthy alteration, other updates consisted mainly of tweaking functionalities and ironing out small bugs.

With a growing amount of D3.js users, the development kept going forward as well. In August 2011 the first major version change was released: v2.0 [13]. It addressed the users' need for more concise code by eliminating code duplication, additionally it introduced selector functions, transparent transitions, prototype injection and a lot of other incremental improvements. Version 2 was also the version in which a testing suite was introduced to the library with the goal of guaranteeing the correctness of the core functionality. The addition of a testing suite is a clear example of an adaptation in response to a need of providing a stable and predictable library to the growing D3.js community. The release included 1,200+ tests covering 90% of the library's functionality.

From that point onward it took another 1,5 year before the next major version v3.0 [14] was released on December 2012. This release saw the introduction of a powerful new geographic projection system, new plugins and improved transitions. Furthermore, a whole host of performance and bug fixes were released. The main focus seemed to be on making it easier for developer to use the library. One change that should be mentioned is the so called chaining of transition functions. Instead of listening to an "end" event, the next function would automatically be applied. The evolutional need addressed in this release, was the developers' need for the library to keep up with smart and easy ways of using the library.

Future release

D3.js is currently undergoing a restructuring and refactoring phase of the entire codebase which will most likely be released this year as version 4.0. The new release won't introduce much in the way of new functionality, but will instead focus on making the library much more modular and easier to maintain. In its current form, D3.js is massive monolithic library which users have to include in its entirety, even if they need only a small part of the library's functionality. Version 4.0 seeks to address this by making everything a module with little to no dependencies. Because the whole library has become a set of low-dependency modules, maintenance and contributing has also become much easier. If a bug arises in some module, that bug will only affect that specific module and won't interfere with the functioning of the other parts of the library.

The need for easier to maintain, low-dependency modules is a clear evolutional need. As can be seen with other JavaScript libraries, there is a shift to making everything an exhaustively tested independent module. Furthermore, by switching to the node package manager (NPM) as its main source of distribution, developers can pick and choose only the functionality they desire, with little to no overhead. With the use of NPM it is also becomes easier to create plugins and add-ons that provide a certain visualization functionality. This specific module will have build in dependencies to D3.js modules, making it much easier for developer who only want to include certain common visualizations.

Trade-offs

While version 4.0 will see a lot of improvements, there are also some possible to downsides to this modularization. For newcomers to D3.js it will probably be a lot more confusing as to what modules they need in order to create visualizations. Furthermore, some modules which have been created are rather low on functionality, which brings up the question if the library was not split in too much modules.

Conclusion

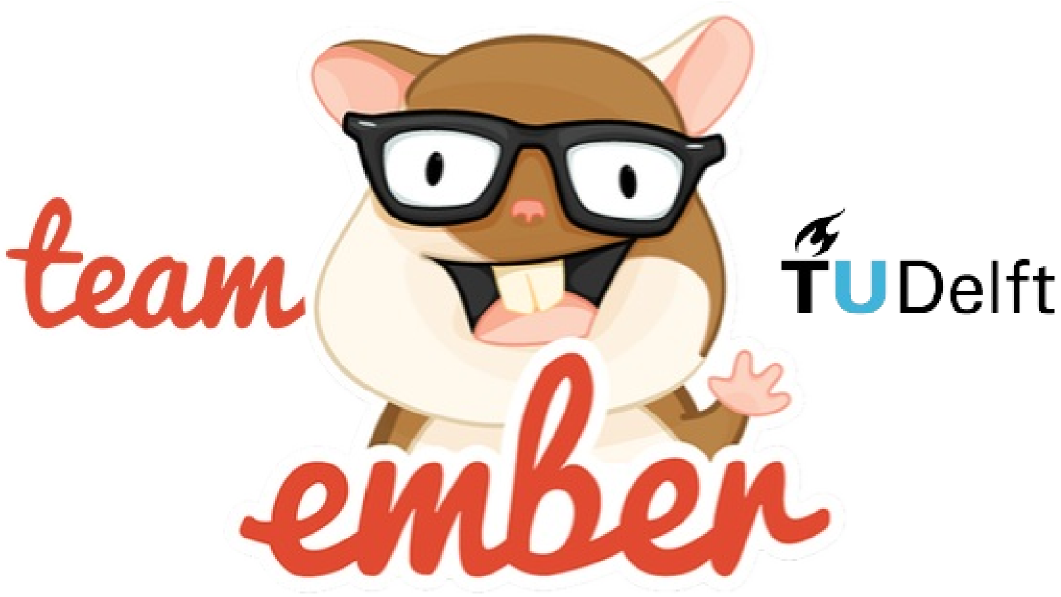
This chapter has hopefully shown that D3.js provides a wide range of tools for not only data visualization, but also for complex filtering and creating user interactions. This makes D3.js to a lot of developers a powerful and useful library suited for a wide range of applications. Besides the functionality, also the library's stakeholders, background, architecture and evolution were discussed. From these discussions it can be concluded that D3.js uses a solid architectural model which is also constantly being improved upon. Finally, the evolution and future plans showed that D3.js is ready for the future. As such, our final conclusion on D3.js is that it is a well-built and decently constructed library which is exceptionally well-suited for what it is created for.

References

1. Dhanji R Prasanna. Dependency injection. Manning Publications Co., 2009.
2. M. Bostock, "Hi! I’m Mike Bostock, creator of D3.js and a former graphics editor for The New York Times", *Reddit*, 8 September 2015. [Online]. Available: https://www.reddit.com/r/dataisbeautiful/comments/3k3if4/hi_im_mike_bostock_creator_of_d3js_and_a_former/. Accessed on: 20 March 2016
3. Multiple (mainly Shawn Allen, Chris Viau), "What companies are using d3.js in production?", *Quora*, 24 February 2012. [Online]. Available: https://www.quora.com/What-companies-are-using-d3-js-in-production. Accessed on: 31 March 2016
4. Mike Bostock, "d3", *NPM*, February 2016. [Online]. Available: https://www.npmjs.com/package/d3. Accessed on: 31 March 2016
5. Online discussion - users of D3.js, "D3JS to visualizae BIG DATA", *Google groups*, 18 April 2013. [Online]. Available: https://groups.google.com/forum/#!topic/d3-js/aRKFtUaE5h4. Accessed on: 17 March 2016
6. Peter May, "Companies using d3.js?", *Data Discourse - Experfy*, 18 November 14. [Online]. Available: https://data.experfy.com/t/companies-using-d3-js/296. Accessed on: 31 March 2016
7. M. Bostock, "D3.js - Data-Driven Documents", *d3.js*, 2015. [Online]. Available: https://d3js.org. Accessed on: 30 March 2016

8. Liam Andrew, "Mike Bostock wants us to visualize algorithms, not just the data that feeds into them", *NiemanLab*, 26 June 2014. [Online]. Available: http://www.niemanlab.org/2014/06/mike-bostock-wants-us-to-visualize-algorithms-not-just-the-data-that-feeds-into-them/. Accessed on: 31 March 2016
9. B. Fry. Visualizing data. Beijing: O'Reilly Media, Inc., 2008.
10. Nick Rozanski and Eoin Woods. Software Systems Architecture: Working with Stakeholders using Viewpoints and Perspectives. Addison-Wesley, 2012.
11. Google, "Closure Compiler", *Google Developers*, 13 February 2016. [Online]. Available: https://developers.google.com/closure/compiler/. Accessed on: 29 March 2016
12. Jason Davies, "Replace Google's closure compiler with UglifyJS.", *github/mbostock/d3*, 25 March 2011. [Online]. Available: https://github.com/mbostock/d3/commit/1dfd3350dcd78ce29f76f4f15da0f94ca219ffad. Accessed on: 31 March 2016
13. Mike Bostock, "2.0 Enter and Update", *github/mbostock/d3*, 23 August 2011. [Online]. Available: https://github.com/mbostock/d3/releases/tag/v2.0.0. Accessed on: 29 March 2016
14. Mike Bostock, "3.0 Baja", *github/mbostock/d3*, 21 December 2012. [Online]. Available: https://github.com/mbostock/d3/releases/tag/v3.0.0. Accessed on: 30 March 2016
15. Luke Dormehl, "The Five Best Libraries For Building Data Visualizations", *Fast Company*, 28 April 2014. [Online]. Available: http://www.fastcompany.com/3029760/the-five-best-libraries-for-building-data-vizualizations. Accessed on: 31 March 2016
16. Nishith Sharma, "The 14 best data visualization tools", *TheNextWeb*, April 2015. [Online]. Available: http://thenextweb.com/dd/2015/04/21/the-14-best-data-visualization-tools/#gref. Accessed on: 30 March 2016
17. Mike Bostock, "D3 Wiki Gallery", *github/mbostock/d3*. [Online]. Available: https://github.com/mbostock/d3/wiki/Gallery. Accessed on: 30 March 2016

Ember: A framework for creating ambitious web applications



Andri Rahmadhani, **Jesse Kaptein**, **Sambit Praharaj** and **Shashwat Verma**

Delft University of Technology, 2016

Abstract

Ember, the brainchild of Yehuda Katz (@wycats), is maintained by a large community of developers and consumers always striving for novelty and simplicity. It allows developers to build scalable web applications and is used by major websites such as Yahoo! and LinkedIn. Currently, it is run by thousands of contributing members from all over the world. This chapter will describe the Ember framework according to multiple perspectives, including the context view, which describes the interaction with its environment. The development view, then, describes Ember's different modules and layers. Subsequently, Ember's features and functionalities are discussed in the feature list and deployment view. Finally, the evolution perspective carves the roadmap for future releases.

Table of Contents

Introduction: Starting the Expedition

Ember.js is an open-source JavaScript web framework, based on the Model-View-Controller (MVC) pattern. It allows developers to create scalable web applications by incorporating common idioms and best practices into the framework. Ember.js is used on many popular websites, including Yahoo! and LinkedIn. Although primarily considered a framework for the web, it is also possible to build desktop and mobile applications in Ember. The best-known example of this is Apple Music. Note that Ember.js is only one product of a complete front end stack built and supported by Ember. Examples of other products are *Ember Data*, *Ember Inspector* and *Liquid fire*.

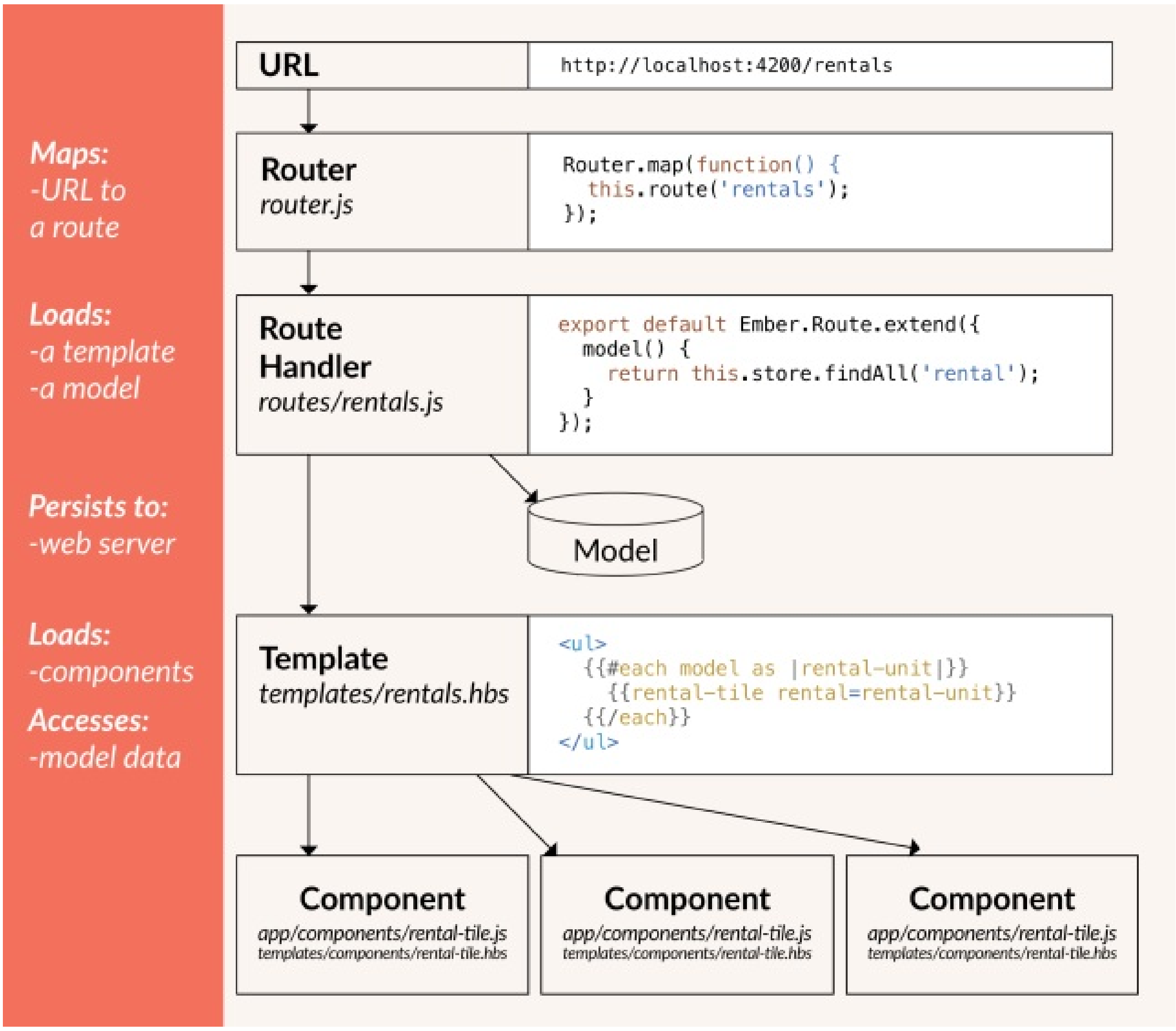
Over the past few weeks, team-Ember, from the Delft Students on Software Architecture (DESOSA)-group, have dived into the framework by stepping through its various components from different perspectives and views. With these insights, we hope to contribute to the framework by both solving issues and make it accessible for readers to contribute too. These issues can vary from bugs to potential new features. In the end, we will bring all the different views and perspectives together to examine whether Ember's architecture is robust or if particular components have to be improved or extended. Before doing so, first, we will guide the reader through Ember's basics.

Ember Basics: Getting Started with the Framework

As described, Ember is built for developers to create web applications effectively and efficiently. However, before one can start writing any Ember code, it is a good idea to get an overview of how an Ember.js application works. The framework itself consists of a couple of basic concepts, namely [1]:

- **Router:** The router translates a URL into series of templates, each backed by a model. As the templates or models change, Ember automatically keeps the URL in the browser's address bar up-to-date. This means that a developer is always able to modify its application, without disturbing the user that much;
- **Route (handler):** A route is an object that tells the template which model it should display. This is simply done by linking user events (e.g. mouse click) to other templates;
- **Models:** A model is an object that stores persistent state. Templates are responsible for displaying the model to the user by turning it into HTML;
- **Templates:** A template describes the user interface. Each template is backed by a model, and the template automatically updates itself if the model changes;
- **Components:** A component is a custom HTML tag. They allow developers to create reusable controls that can simplify other templates;

Figure 1 presents the described relations between the router, routes, models, templates and components [1].


Maps:
-URL to
a route
Loads:
-a template
-a model
Persists to:
-web server
Loads:
-components
Accesses:
-model data
URL
http://localhost:4200/rentals
Router
router.js
Route
Handler
routes/rentals.js
Model
Template
templates/rentals.hbs
Component
app/components/rental-tile.js
templates/components/rental-tile.hbs
Component
app/components/rental-tile.js
templates/components/rental-tile.hbs
Component
app/components/rental-tile.js
templates/components/rental-tile.hbs

```
Router.map(function() {
  this.route('rentals');
});
```

```
export default Ember.Route.extend({
  model() {
    return this.store.findAll('rental');
  }
});
```

```
<ul>
  {{#each model as |rental-unit|}}
    {{rental-tile rental=rental-unit}}
  {{/each}}
</ul>
```

Figure 1: The way in which Ember's core concepts interrelate with each other

Stakeholders: The Team behind Ember

The Ember society consists of a large community of thousands of software developers, architects, users, and testers, with most of them having another job and working voluntarily on the Ember.js project. The search for detecting the most relevant stakeholders led immediately to Ember's Core Team Members. It appeared that only a limited group of Core Members were actively involved in approving or denying potential contributions.

For a detailed presentation of the community behind Ember.js, see Figure 2. For a clear interpretation and explanation of terms, please consider Rozanski & Woods (2011).

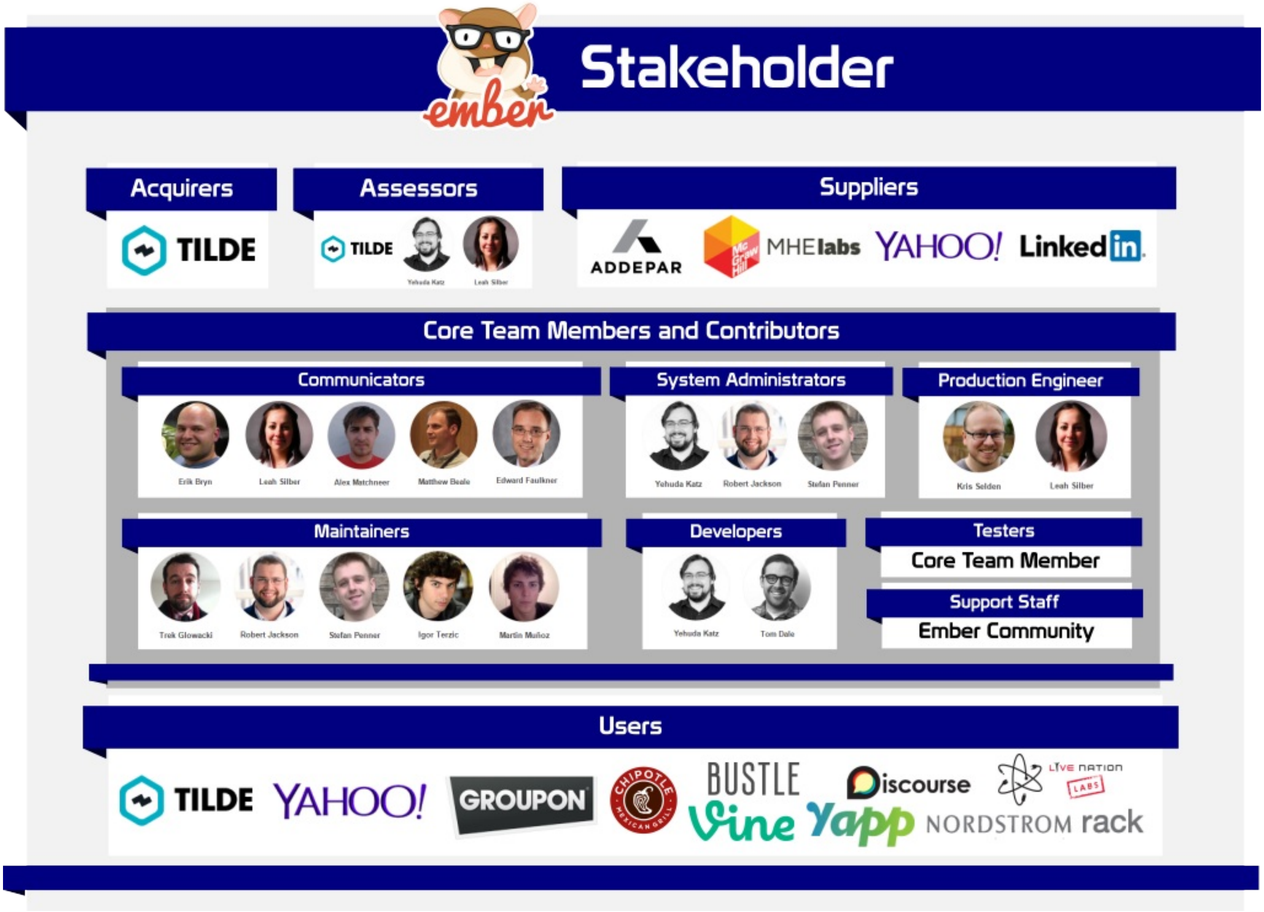


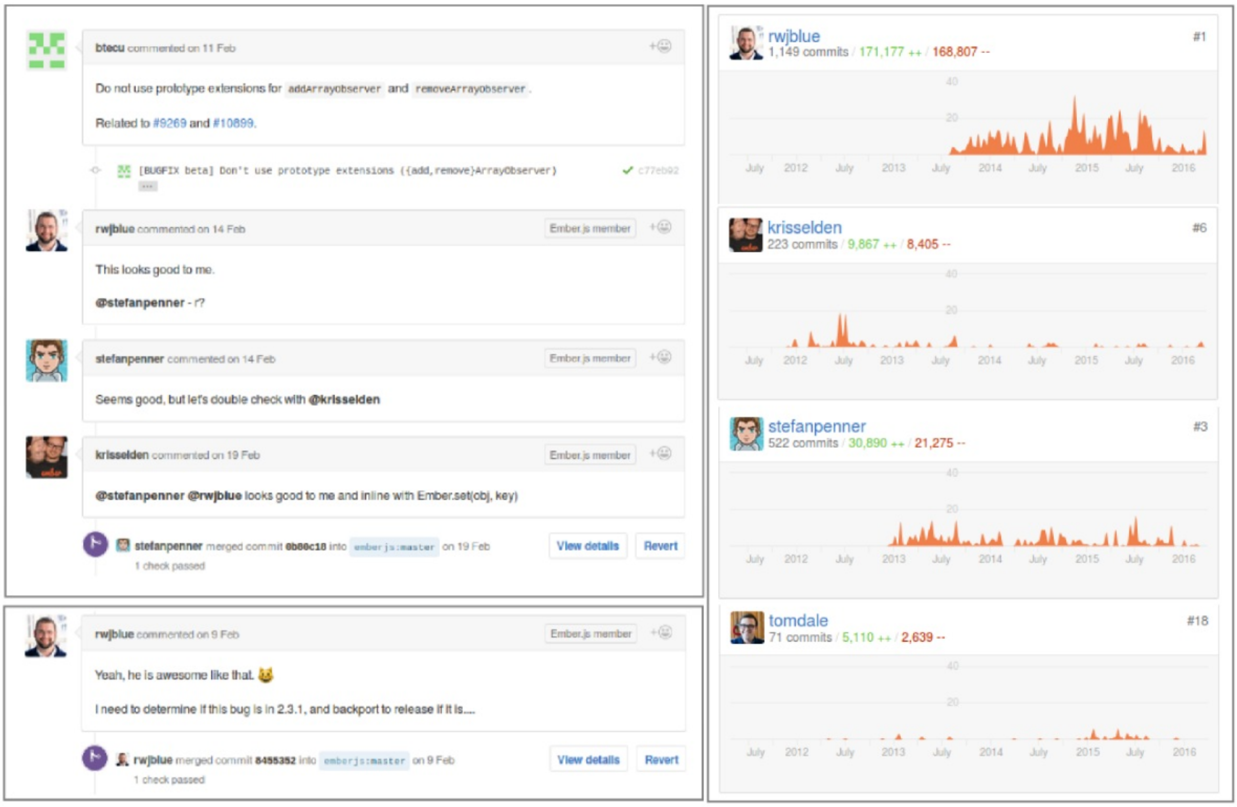
Figure 2: An oversight of all relevant stakeholders in the Ember framework and their roles

Core Team Members and Contributors

The core team behind Ember.js is relatively small and, therefore, they are considered to be both developers, maintainers, testers, communicators and assessors within the community. Currently, the core team consists of thirteen members, of which we will describe the most important:

- **Yehuda Katz** (@wycats), founding father of the Ember framework and still mainly responsible as a developer and system administrator;
- **Robert Jackson** (@rwjblue) & **Stefan Penner** (@stefanpenner), both one of the most active members in the Ember.js GitHub repository;
- **Tom Dale** (@tomdale), important developer and tester of the Ember framework and again there from the start in 2011;
- **Kris Selden** (@krisselden), important production engineer of the Ember framework.

Based on GitHub activities as seen in Figure 3, one can conclude that mainly *Robert Jackson*, *Stefan Penner*, *Tom Dale* and *Kris Selden* are involved in approving or denying contributions to the Ember framework.



btecu commented on 11 Feb

Do not use prototype extensions for `addArrayObserver` and `removeArrayObserver`.

Related to #9269 and #10899.

`[BUGFIX beta] Don't use prototype extensions ({add,remove}ArrayObserver)` c77eb92

rwjblue commented on 14 Feb — Ember.js member

This looks good to me.

@stefanpenner - r?

stefanpenner commented on 14 Feb — Ember.js member

Seems good, but let's double check with **@krisselden**

krisselden commented on 19 Feb — Ember.js member

@stefanpenner **@rwjblue** looks good to me and inline with Ember.set(obj, key)

stefanpenner merged commit **0b80c18** into `emberjs:master` on 19 Feb
1 check passed

rwjblue commented on 9 Feb — Ember.js member

Yeah, he is awesome like that. 😺

I need to determine if this bug is in 2.3.1, and backport to release if it is....

rwjblue merged commit **8455352** into `emberjs:master` on 9 Feb
1 check passed

rwjblue #1
1,149 commits / 171,177 ++ / 168,807 --

krisselden #6
223 commits / 9,867 ++ / 8,405 --

stefanpenner #3
522 commits / 30,890 ++ / 21,275 --

tomdale #18
71 commits / 5,110 ++ / 2,639 --

Figure 3: Ember.js Github activities

Users & Suppliers

Ember is used on many popular websites, including Yahoo! and LinkedIn. Some of these larger users support the framework financially. Other, smaller users, can help improve the framework by reporting bugs, performance issues or proposing new features. Therefore, many of the fixed bugs are dedicated to users. Furthermore, small users can also make donations to the project, buy unique Ember.js gear or participate in Ember.js courses and conferences, to support future development of the framework.

Context View: Interacting with Ember's environment

By developing a broader view, the frameworks relationships, dependencies, and interactions with other systems and parties will appear. In Figure 4 such a context diagram is presented. In this figure, we see that Ember.js is (technically) based on a couple of software packages and languages. To install and configure Ember.js, one should have *Node.js* and *npm package manager*. To build templates and make use of Ember's vast library, though, understanding of both *Handlebars* and *jQuery* are required. For creating modules, the ES6 module standards are used, whilst the technical core is based on *JavaScript Model-View-Controller* pattern in combination with *HTML*.

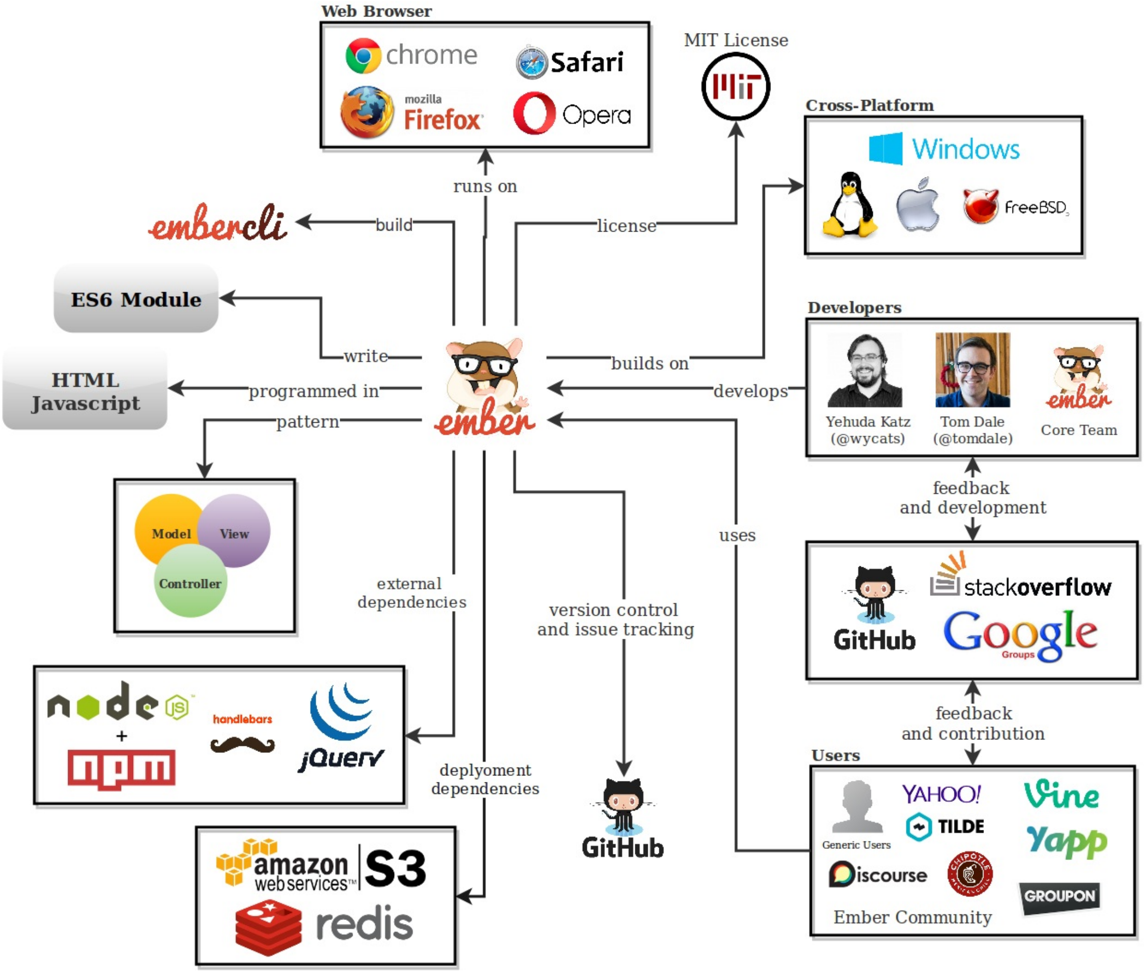


Figure 4: Ember.js context view diagram

To maintain and help the large Ember community, a couple of third party platforms are used. For user support, Ember uses StackOverflow. Other channels for user support can be found in the Ember Community on Slack, the IRC channel on Freenode IRC or within Google Groups. For Ember learning resources (podcasts, videos, blog posts, books and so on) almost all Core Team Members have public channels on either YouTube or Vimeo. Most of these platforms are also able to give feedback for future development. Finally, for developers, Ember's source code is, of course, available on GitHub.

Lastly, the figure shows that Ember applications can be built on a cross-platform server running Node.js, such as Windows, OSX, Linux, and FreeBSD. The framework can be rendered on multiple web browsers like Google Chrome, Firefox, Opera, and Safari. Also, Ember uses an efficient deployment technique using Redis and Amazon S3, which will be discussed in the later section.

Development View: The Foundations of the Ember.js

As a framework, Ember.js makes use of the Model-View-Controller (MVC) design pattern for the users to build their applications. MVC is used for relating user interface to the underlying model, providing several advantages, such as [2]: clean separation of concerns, simplified testing, improved scalability, easier maintenance, reuse of code and decoupling of application layers. The different modules and components of Ember's framework are presented in Figure 5 [1].

This section starts discussing the modular design that forms the foundations of Ember.js to gain a deeper understanding of Ember's architecture. Furthermore, some standardization aspects and Ember's codeline organization are considered.

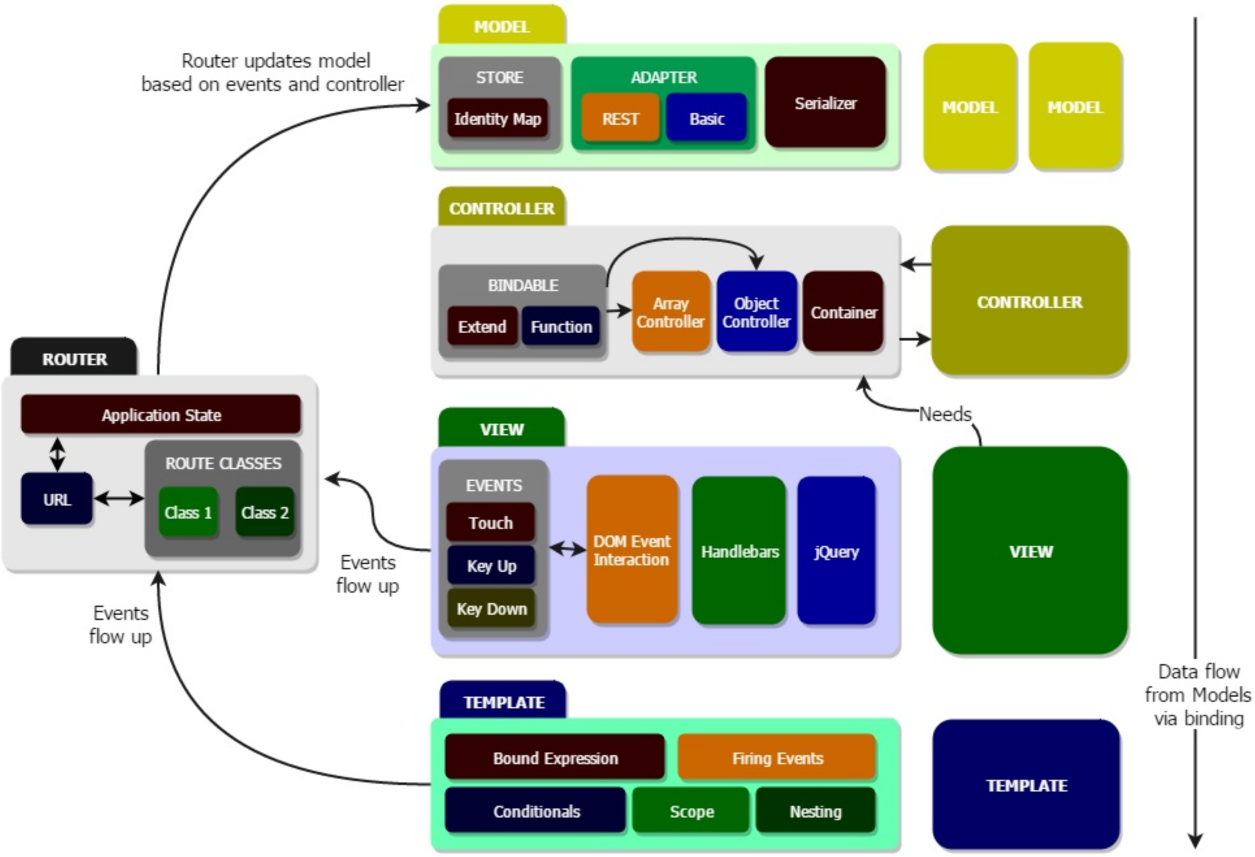


Figure 5: Architectural View of MVC Design Pattern Provided by Ember.js

Modular Design: Ember's MVC Design Pattern

Firstly, in Figure 4, the yellow block describes Ember's **models**. Models represent the underlying data that an application presents to the user. There is no doubt that models can vary heavily through different apps. A photo sharing application might have, for example, a `Photo` model to show a specific photo, whilst a `PhotoAlbum` represents a group of pictures. Contrary, a shopping website would probably have other models, such as `ShoppingBag` or `Payment`. Note that each model testable in isolation and can easily be changed or replaced.

They work with the router by a `find(id)` command. Thus, each module has its identity (id) which is inside its store. The REST adapter loads and saves the various records. As presented, data flows from models to templates in top-down fashion via bindings.

Secondly, **controllers** are shown. In recent versions of Ember.js, controllers are being replaced by **components**, making them less important in the framework. The controller module is only being used to pass user actions through the different layers when moving from components to routes. Note that they are also testable in isolation. Controllers consist of object controllers and array controllers. Array controller contain lists of elements and also interact with object controller.

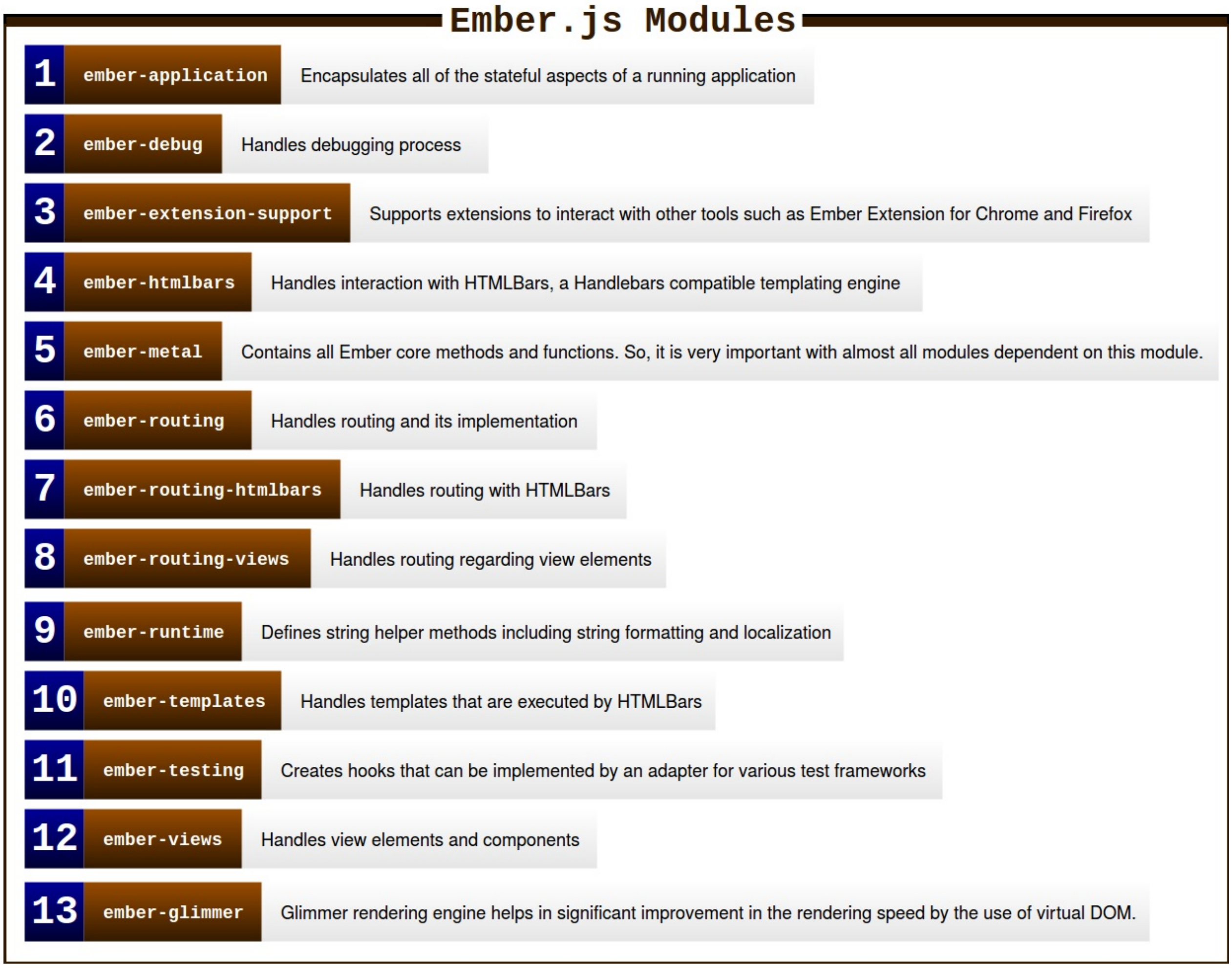
Thirdly, **views** render HTML elements and manage the user interface. In this, DOM (Document Object Model) helps with event interactions, like mouse overs and key ups. Handlebars are just like HTML bars which are self-updating **templates**. Note that controllers need views to work.

Fourthly, **templates** are more or fewer components of the view module. They more or less boost the app's user interface. They consist of the bound expressions, firing events, conditions, scope of the code and nesting (like nested loops). It helps in firing events from the view to the router. Thus, it defines the basic building blocks for the app's (front-end) structure.

Finally, **routers** update the model based on the templates (events) and the controller. It consists of the application state, URL, and different route classes. The URL interacts with the application state and the route classes to forward the events to the model.

Module Organization: Dependencies and Interactions

As a complex system, Ember.js source code can be classified into several modules as seen in Figure 6. Whilst Ember's design pattern for developers is modular and easy to interpret, the application's source code is not.



Ember.js Modules

1	ember-application	Encapsulates all of the stateful aspects of a running application
2	ember-debug	Handles debugging process
3	ember-extension-support	Supports extensions to interact with other tools such as Ember Extension for Chrome and Firefox
4	ember-htmlbars	Handles interaction with HTMLBars, a Handlebars compatible templating engine
5	ember-metal	Contains all Ember core methods and functions. So, it is very important with almost all modules dependent on this module.
6	ember-routing	Handles routing and its implementation
7	ember-routing-htmlbars	Handles routing with HTMLBars
8	ember-routing-views	Handles routing regarding view elements
9	ember-runtime	Defines string helper methods including string formatting and localization
10	ember-templates	Handles templates that are executed by HTMLBars
11	ember-testing	Creates hooks that can be implemented by an adapter for various test frameworks
12	ember-views	Handles view elements and components
13	ember-glimmer	Glimmer rendering engine helps in significant improvement in the rendering speed by the use of virtual DOM.

Figure 6: Ember.js modules

It is difficult to group and partition modules in Ember.js into different layers as they form a complex dependency pattern (see Figure 7; this figure is constructed by visualizing Ember.js's source code through GraphViz). Some of these modules interact with third party modules as well, making the interaction pattern even more complex. The relatively clear defined modules prevent develop functions multiple times and make it easy to localize bugs and performance issues, though. However, the dependencies between the different modules make it hard to change specific components, because it can be hard to predict its consequences for other modules. Subsequently, this has a negative impact on both scalability and maintainability of the Ember.js framework. This is in particular true for *ember-metal*, *ember-runtime* and *ember-views*.

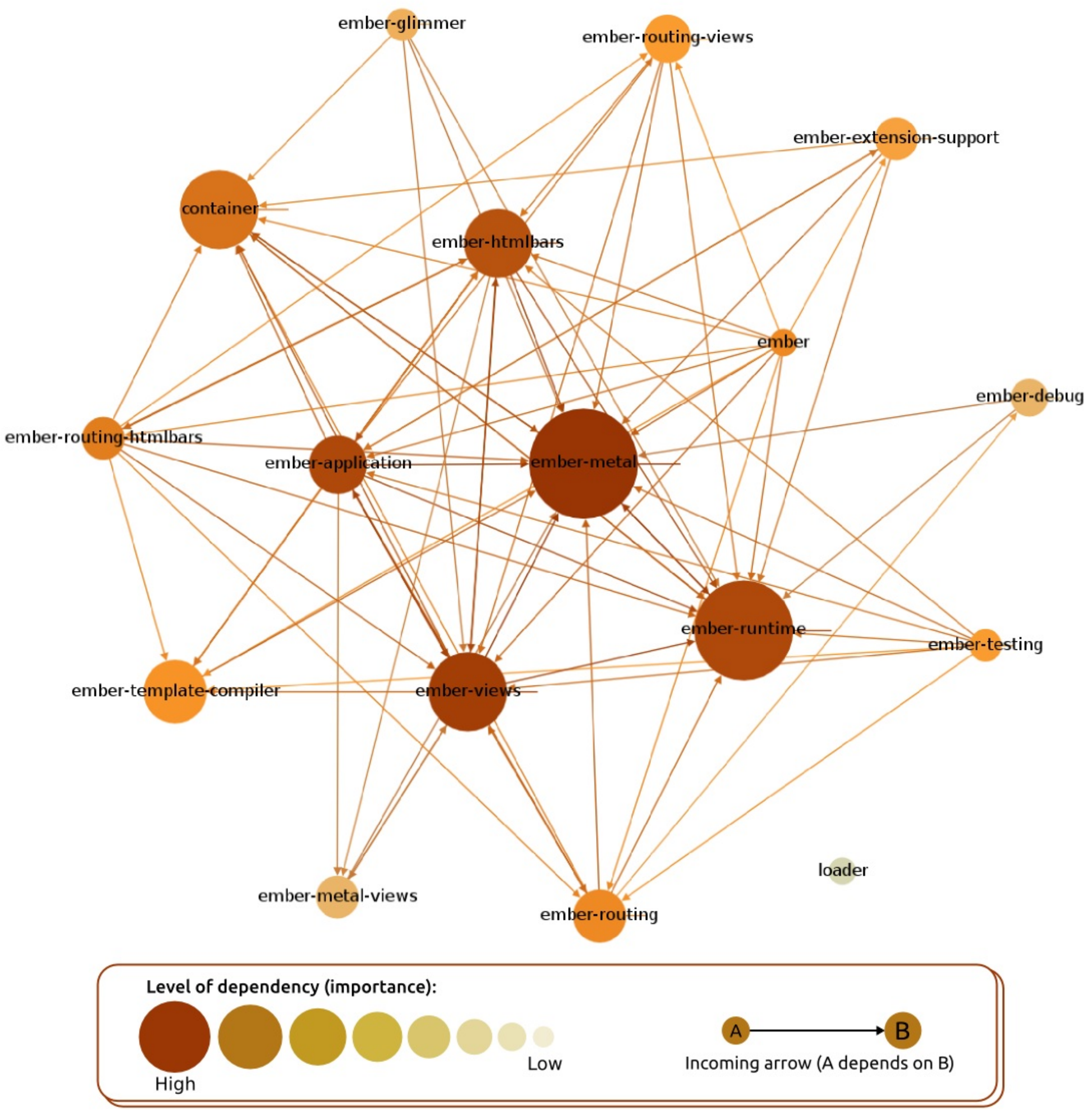


Figure 7: Ember.js modules and their dependencies

Standardization of Design and Testing

Ember's internals and most of the code we will write in our applications takes place in a run loop. The core codebase of Ember uses ES6 modules which help in improving the runtime i.e. if we have things set up properly in our development environment, then we can import just the parts of Ember.js that we want to use.

For developers, Ember team provides JavaScript style guide, which gives best practices for new contributors and developers. It makes the code more readable and maintainable. Also, Ember team provides a guideline for developers and contributors to add new features through an RFC process. Ember uses Configuration to configure dependencies and other things for developing Ember.js.

Every developer and contributor should perform a unit test, typically using Qunit, before submitting a PR (Pull Request) which potentially changes parts of the code. Ember.js implements a continuous integration platform called Travis CI. It is used to test each PR before it is merged. When developers submit a modification of code, Travis will automatically be launched and a note will be added to the PR.

Codeline Organization

The term codeline is mainly used to describe the arrangement of the source code files. Ember's codeline can be organized into a well-defined structure as shown in Figure 8.

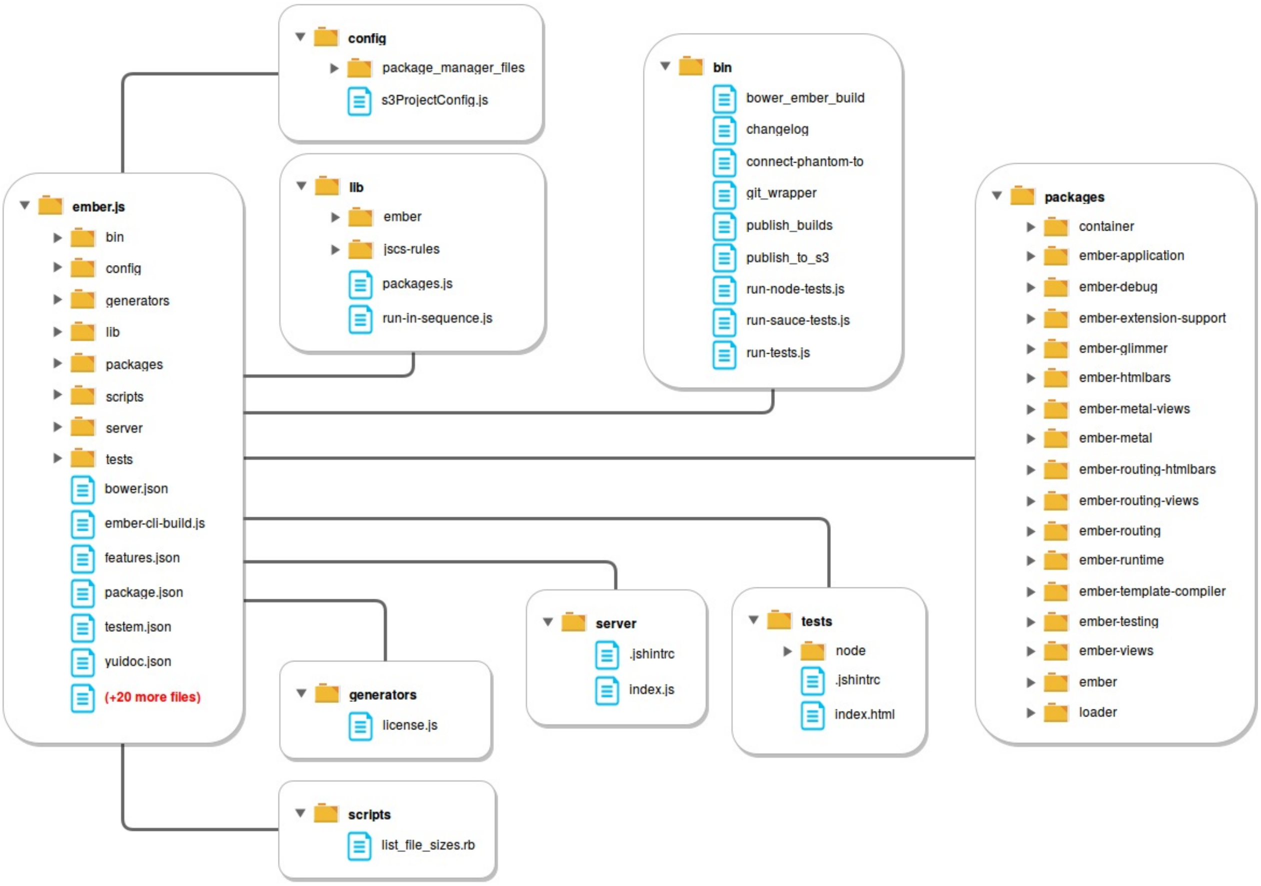


Figure 8: Directory structure

The Ember.js directory consists of main folders like *bin*, *ember.js* and *packages*. In the root directory, `bower.json` and `ember-cli-build.js` are the most important files. `bower.json` is a Bower package manager file which helps managing dependencies, whilst `ember-cli-build.js` is the build configuration to be used by `ember-cli`. In *bin* folder, the `changelog` keeps track of the changes in version and the commits. *Packages* consist of the main modules which have been described in the Ember's Module Organization. The *config* folder contains the configuration files that handle the mapping of a specific configuration with corresponding files. *Generators* contain the `license` and *lib* consists of the configurations of packages.

Features from different Perspectives

This section provides a generic insight in the main features of the Ember.js framework along with the perspective of the different stakeholders (see Figure 2). We have seen that there are two important stakeholder groups, namely the actual users of the framework and its developers team. It appears that Ember.js has dozens of features, mainly for their users. Here, a couple of relevant features for users and one important feature for both users and developers are discussed.

Users: Data Model, Routing, Rendering Engine & Configuration of Application

Ember provides a robust **data model** using Ember Data. It retrieves models from the server as JSON, save updates to the server and create new models in browsers. Ember Data can be configured to connect with different types of backends without writing any networking code by accomodating a relevant adapter provided in the Ember Data Adapters collection. Users can easily extend the default Adapter if they work with non-standard backend using `DS.JSONAPIAdapter` .

Secondly, Ember uses **routers** as the URL routing mechanism. It has four options, namely: `history` (uses the HTML5 History API), `hash` (uses anchor-based URLs), `auto` (uses history if supported by the user's browser and falls back to hash otherwise) and `none` (doesn't update the URL). This option can be configured by users in `config/environment.js` under `ENV.locationType` . Ember uses also *controllers* that allow developers to decorate models with display logic. Ember Controller has two different types: either `automatic` or `custom` . They can be selected and configured using `ember-cli` . The `custom` controller provides flexibility to users to extend the behavior of one of the automatically generated controller.

Thirdly, Glimmer is used as the latest *rendering engine* of Ember.js. It differentiates between static and dynamic components, because of expressiveness of Handlebar's templates [3]. Glimmer can be configured through feature list during the build process.

Finally, Ember users can **configure their application** in a configuration file located in `config/environment` . This is a default file generated by Ember CLI. Here, the users can change the default environment by defining `ENV` object, which is currently limited to three: development, test and production [1].

To conclude, users can benefit from Ember’s flexibility by choosing between three different environments. The users can disable and enable features in each of environments depending on requirements they have. Ember users can configure Ember CLI by adding configuration codes to the `.ember-cli` file in the application root.

Developers & Users: Configuration of New Features

Ember uses **Features Flag** that manages Ember.js features used in a project. Ember users can enable or disable the features listed in `features.json` file, which is located in the root of the Ember.js project, by changing the application configuration file as stated in the previous section. The `features.json` file displays all new features and their current status. A feature can have one of a three flags [1]:

- `true` - The feature is present and enabled: the code behind the flag is always enabled in the generated build.
- `null` - The feature is present but disabled in the build output. It must be enabled at runtime.
- `false` - The feature is entirely disabled: the code behind the flag is not present in the generated build.

The description of newly-flagged features can be found in `FEATURES.md` file located in *master* branch on Github. In this file, developers explain what certain features do and the link to the original pull request is given. This, so that users get knowledge regarding how the feature is implemented in Ember.js.

Newly-flagged features have a Feature Life-Cycle, starting from Canary, Beta, to Release phase. In Beta phase, Ember's Core Team will evaluate each of new features that come from Canary phase. When the features are considered to be stable, they are made available in the next beta phase and enabled by default. As the beta cycle completed, the features will available in the release phase and becomes part of Ember framework [1].

An example of a newly-flagged feature is Glimmer rendering engine, which is still in Beta phase. Glimmer potentially is a significant improvement to the Ember.js framework, as can be seen in pull request #10501. For this reason, this feature is flagged as `null` in `features.json` file.

Deployment View: The Operational Environment

Considering the vast use of Ember.js, it is important to describe the deployment of software to guarantee its operation in different environments, as defined in Rozanski & Woods [4]. The deployment viewpoint applies to any system with a required deployment environment that is not immediately obvious to all of the interested stakeholders.

Third Party Software Requirements

Identifying third-party software requirements are essential for both developers and users. For developers, requirements make it clear what tools or libraries to use. On the other hand, for users can understand what is needed to apply software in their environment. Fortunately, as Ember.js works with JavaScript, it requires only a handful of third party tools.

Each software element in Ember.js imposes requirements for specific versions of third party software. Most of these require separate installation as they do not come in one, complete package.

For Ember.js, the following tools are required for configuration:

- **npm**: This manages dependencies for an application running on it. It is also written in JavaScript and is maintained via GitHub. Required version v2.4+;
- **Node.js**: This is an open-source, cross-platform runtime environment for developing server-side web applications [5]. Required version v4.x+;
- **Bower**: Bower is a package management system for client/developer side programming on the internet. It depends on Node.js and npm. It has to be v1.0+;
- **Browser support**: After v2.0 Ember does not support IE8 or below to render its applications.

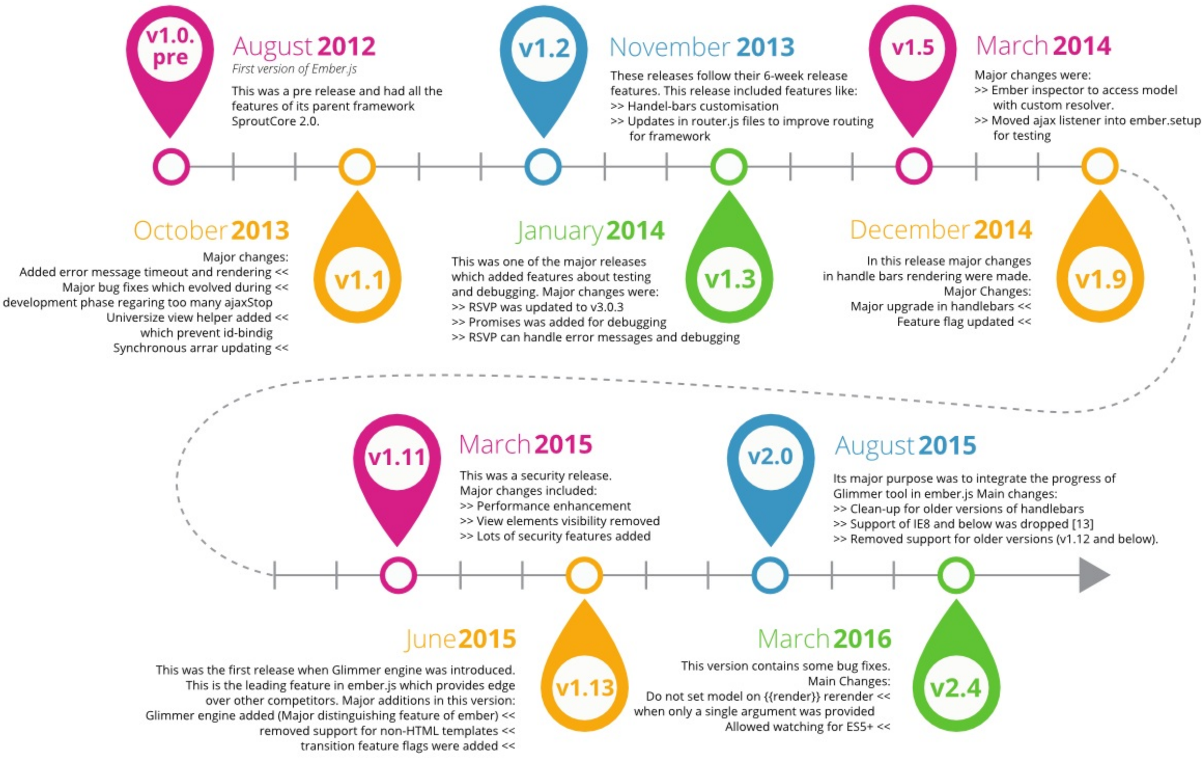
These version requirements are not related to Ember.js, but we found them out when building it in our environment.

Network Requirements

Ember.js runs a state manager and reverse proxy on the web server. The node in the system can be divided into two categories: server and client. On the server side, Ember.js has limits on version issue which get deployed after a certain version has passed deployment tests. These limits are in place to protect services from both accidental and untimely crashes. On the client side, the network requirements are not specified.

Evolution: How has Ember become the framework it is today?

To deal with the variability of an ever changing environment, Ember has adopted a six-week release cycle, inspired by the rapid release cycle of Google Chrome [6]. As described, this approach allows newly developed features to be carefully released when the community considers them ready to use. For versioning, Ember uses the semantic versioning convention [6]. It means that new features and fixes for small bugs are added by point releases (1.1, 1.2, ...), whilst breaking changes are only introduced at major version numbers (1.0, 2.0, ...). The complete version history can be seen in Figure 9.



August 2012
First version of Ember.js
This was a pre release and had all the features of its parent framework SproutCore 2.0.

November 2013
These releases follow their 6-week release features. This release included features like:
>> Handel-bars customisation
>> Updates in router.js files to improve routing for framework

March 2014
Major changes were:
>> Ember inspector to access model with custom resolver.
>> Moved ajax listener into ember.setup for testing

October 2013
Major changes:
Added error message timeout and rendering <<
Major bug fixes which evolved during <<
development phase regaring too many ajaxStop
Universize view helper added <<
which prevent id-bindig
Synchronous arrar updating <<

January 2014
This was one of the major releases which added features about testing and debugging. Major changes were:
>> RSVP was updated to v3.0.3
>> Promises was added for debugging
>> RSVP can handle error messages and debugging

December 2014
In this release major changes in handle bars rendering were made.
Major Changes:
Major upgrade in handlebars <<
Feature flag updated <<

March 2015
This was a security release.
Major changes included:
>> Performance enhancement
>> View elements visibility removed
>> Lots of security features added

August 2015
Its major purpose was to integrate the progress of Glimmer tool in ember.js Main changes:
>> Clean-up for older versions of handlebars
>> Support of IE8 and below was dropped [13]
>> Removed support for older versions (v1.12 and below).

June 2015
This was the first release when Glimmer engine was introduced. This is the leading feature in ember.js which provides edge over other competitors. Major additions in this version:
Glimmer engine added (Major distinguishing feature of ember) <<
removed support for non-HTML templates <<
transition feature flags were added <<

March 2016
This version contains some bug fixes.
Main Changes:
Do not set model on {{render}} rerender <<
when only a single argument was provided
Allowed watching for ES5+ <<

Figure 9: Ember.js version history

So, why does Ember change?

In general, the issues and pull requests in GitHub show a couple of reasons why changes in the Ember.js framework are needed. It turns out that major changes in the framework are caused by developments in both its **supporting languages** and **supported web browsers**. Both languages like JavaScript and web browsers keep developing themselves. To remain compatible with those newer versions, Ember.js releases new versions as well.

Note that pull requests for new features are not expected in the GitHub repository because they move to Ember RFC (*Request for Comment*) for discussion purposes. Furthermore, new versions are also released because of new (or improved) documentation and depreciation. It means that for simplicity purposes, not all web browsers, plugins, features or add-ons are supported anymore. Support for Internet Explorer 8, for example, is available up to version 1.13, but is dropped from release 2.0 onwards #11313. For a complete oversight of all changes made to the Ember framework since 2012, please consider the Ember Changelog.

Change management in theory: different variability mechanisms in Ember.js

Dealing with ever changing software is not easy. Therefore, Ember has installed several variability mechanisms to adapt to the changing environment. To denote variability mechanisms, a slightly changed definition derived from Puhlmann et al. (2005) [7] is used:

"Variability mechanisms denote techniques for the derivation of software from software models". Puhlmann et al. also describes some commonly used variability mechanisms, to which will be referred. Note that we only looked at changes at runtime level, which comprises of adjustments done whilst the software product is already implemented. This is in contrary to product time changes, which refer to types of variability that are resolved during the implementation of a software product.

Document changes are the simplest to deal with. Because of Ember's open-source nature, all documentation is publicly accessible. If document changes are needed, one can open a pull request to do so. The "old" documentation module will simply be replaced by a new one, when the request is approved. Puhlmann et al. denote this as *replacement of components*. If new documentation modules are needed because of the adoption of recent features, new documentation is simply added (*addition of components*). Note that "old" documentation is still being saved (see Ember Guide).

Ember.js manages new features in three different ways:

1. If new features are classified as irrelevant to be incorporated in new releases, then the community relies upon third parties to create add-ons to make use of the suggested features. This may be the case if the feature is useful, but only for a limited amount of users. Puhlmann et al. denote this as *delegation*: the functionality of an object can be extended by delegating the calculations the object cannot perform to another object encapsulating the (varying) functionality for performing the respective calculations;
2. If new features can be incorporated by extending some of the modules upon which Ember.js is based, a point release will be launched. Puhlmann et al. refer to this as *extension points of modules*. Due to the modular structure, Ember's framework can easily adopt such changes;
3. If new features can be incorporated by adjusting the modular structure or adding new modules, a new release will be launched. Puhlmann et al. denote this as the *addition of components*.

Compatibility issues are using the same variability mechanisms as described above. If compatibility issues are not important, then Ember relies upon the community to develop add-ons. If the issues are important enough, then they will be incorporated in future releases. Note that bug fixes are being dealt with only the last two mentioned variability mechanisms. Unimportant bug results in the closing of the pull request. Finally, if the framework is going to be deprecated, particular modules will simply be omitted, or parts of the modules will be adjusted. Puhlmann et al. refer this to *omission of components*.

Change management in practice: from one version to another

Ember.js uses an interesting version change technique, namely *S3+ Redis deployment* (also called as *lightning deployment*), which was introduced about a year ago. With this technique, developers can push static assets (version upgrade components) to *Amazon S3* anytime, see Figure 10.

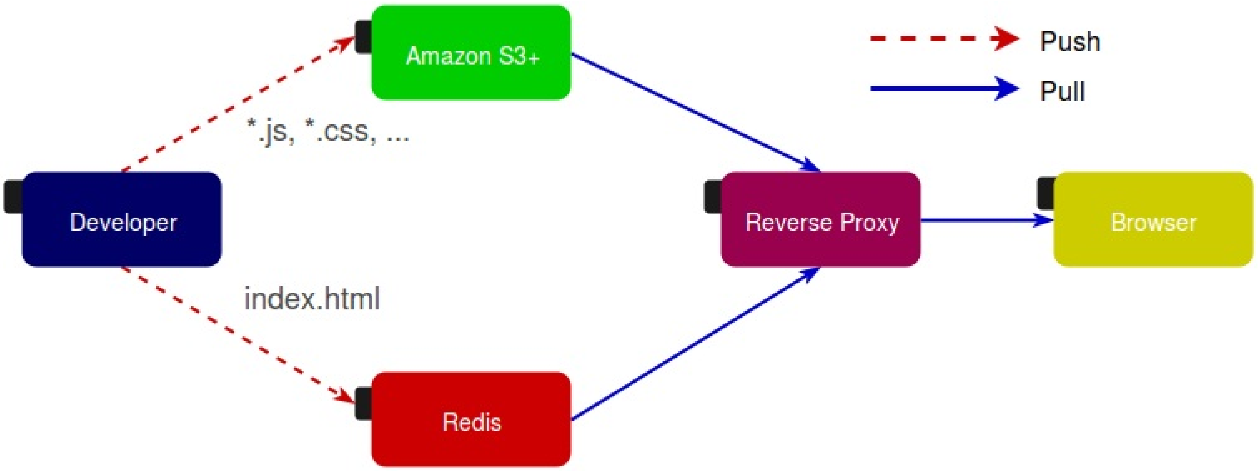


Figure 10: S3+ redis deployment in Ember.js

Amazon S3 allows developers to have different versions of App.js anytime. However, only one of these versions is used in production, which is determined by `index.html` in *Redis*. This allows to release a new version of Ember.js when it is ready to be shipped, and this can be done easily by pushing the corresponding `index.html` into Redis. This new version of `index.html` is recognized by *Reverse proxy*, which picks up the corresponding asset from S3 and transfers the contents to the browser. This deployment technique switches versions lightning fast without any noticeable downtime for users.

Technical Debt: Consequences of System Choices

Developers have roughly two options to implement new features in their applications. One is to do it quick and dirty, which has its impact on future changes. The other option is to do it smart and robust, but this will cost money and time. Technical debt is a metaphor regarding the eventual choice of the implementation and its consequences [8].

Evolution Trade Offs

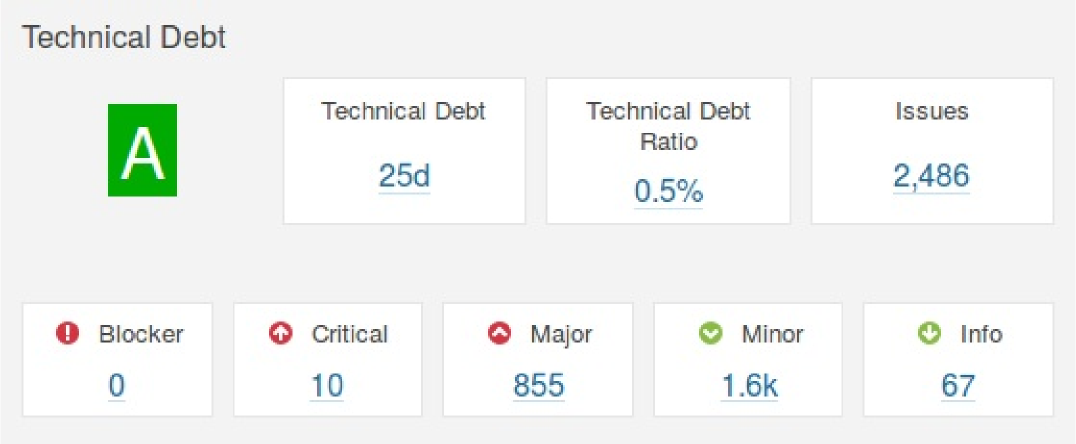
Ever since the development of Ember.js back in 2011, efforts were undertaken to manage and control technical debt of the Ember.js framework. To manage technical debt, for the development team, it was crucial to create a community to develop, build, and test Ember.js applications in an efficient way and make peer reviews possible using pull requests and issues. To do this, they first created Ember App Kit (EAK). EAK is eventually replaced by ember-cli, which is a faster, more user-friendly command line tool that made testing and peers reviewing even more efficient. Even so, technical debt is still present in the Ember framework today.

Ever since the development of Ember.js back in 2011, efforts were undertaken to manage technical debt of the Ember framework. As Stefan Penner called it in his presentation at the 2014 Ember conference: *"We want to build applications that are stable, fast and extensible, oh and, your product feature is due on Tuesday!"* [9]. To manage technical debt, it was crucial to create a community to develop, build, and test Ember.js applications and make peer reviews possible using pull requests and issues. To do this, they first created Ember App Kit (EAK). EAK is eventually replaced by ember-cli, which is a more user-friendly command line tool that made testing and peer reviewing simpler. Even so, technical debt is still present in both the Ember framework today. Stefan Penner also outlined a couple of frequently made trade-offs that has led to technical debt for Ember developers, namely:

- **Coupling & complexity:** to build web applications quickly, Ember uses *Inversion of Control (IoC)*. This means that for the flow of control, a generic, reusable library is used. The (dis)advantage of this is that it hides complexity, which is a good thing for interpretation purposes, but can also lead to tremendous technical debt if things go wrong. To manage this, the Ember framework uses both *containers*, that abstracts away coupling, and *resolvers*, that abstracts the way to find code;
- **Building time & usability:** sometimes, if you have to do things quickly, one may write a single, large code in which loading the code becomes dependent upon the order of lines. *"This is totally crazy"*, according to Penner [9]. To manage this form of technical debt, Ember uses modules based upon ES6 module syntax. This makes testing easier and the application independent of the order of lines.

Technical debt in detail

SonarQube is used to analyze technical debt in the Ember.js framework in more detail. Only the source code inside the `lib` and `packages` folder are analyzed, as these folders contain the core files. SonarQube will raise an issue whenever a code violates certain rules of programming language used, as seen in Figure 11.



Technical Debt

A

Technical Debt	Technical Debt Ratio	Issues
25d	0.5%	2,486

Blocker	Critical	Major	Minor	Info
0	10	855	1.6k	67

Figure 11: Technical debt info generated using SonarQube

Fortunately, no blocking issues were found. Blocking issues are issues with high probability to severely impact the behavior of the application in production. However, there still exists several critical and major issues, mostly about unused variables and functions. The critical issue represents either a bug with a low probability to affect the system or security flaws, whilst major issues deal with quality flaws, which can negatively impact developer productivity [10]. Overall, Ember.js has got a low technical debt ratio, which means that the project is well-maintained. This conclusion is reinforced by the analysis of code duplication, which yields a low percentage of the total code duplication (14.3%). The code duplication for each component is depicted in Figure 12.

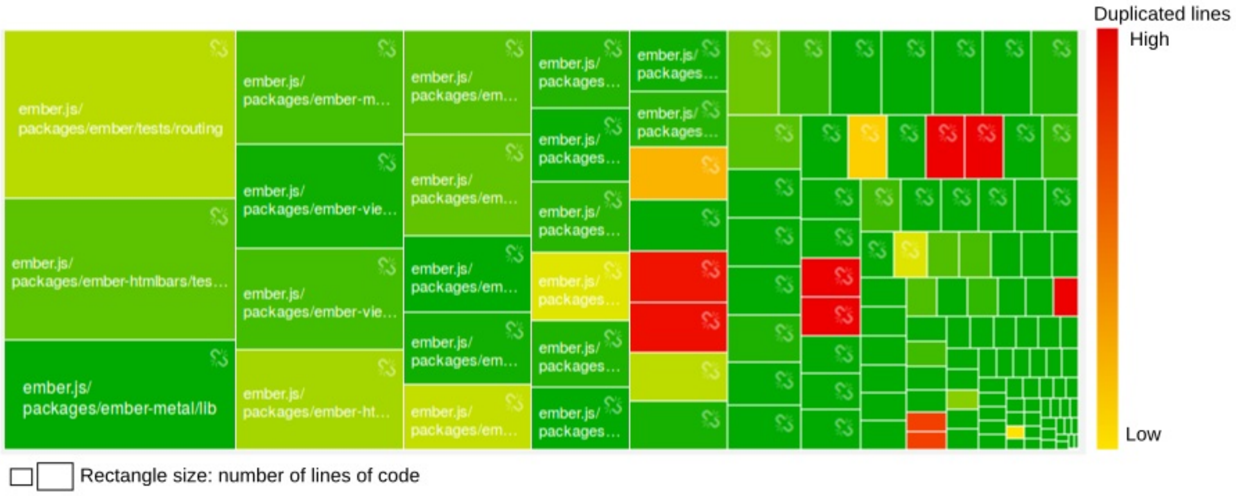


Figure 12: Code duplication portion for each component

Another interesting fact is that relatively most of the issues come from the code that handles routing as presented in Figure 13. Note that the size of the circle represents the technical debt.

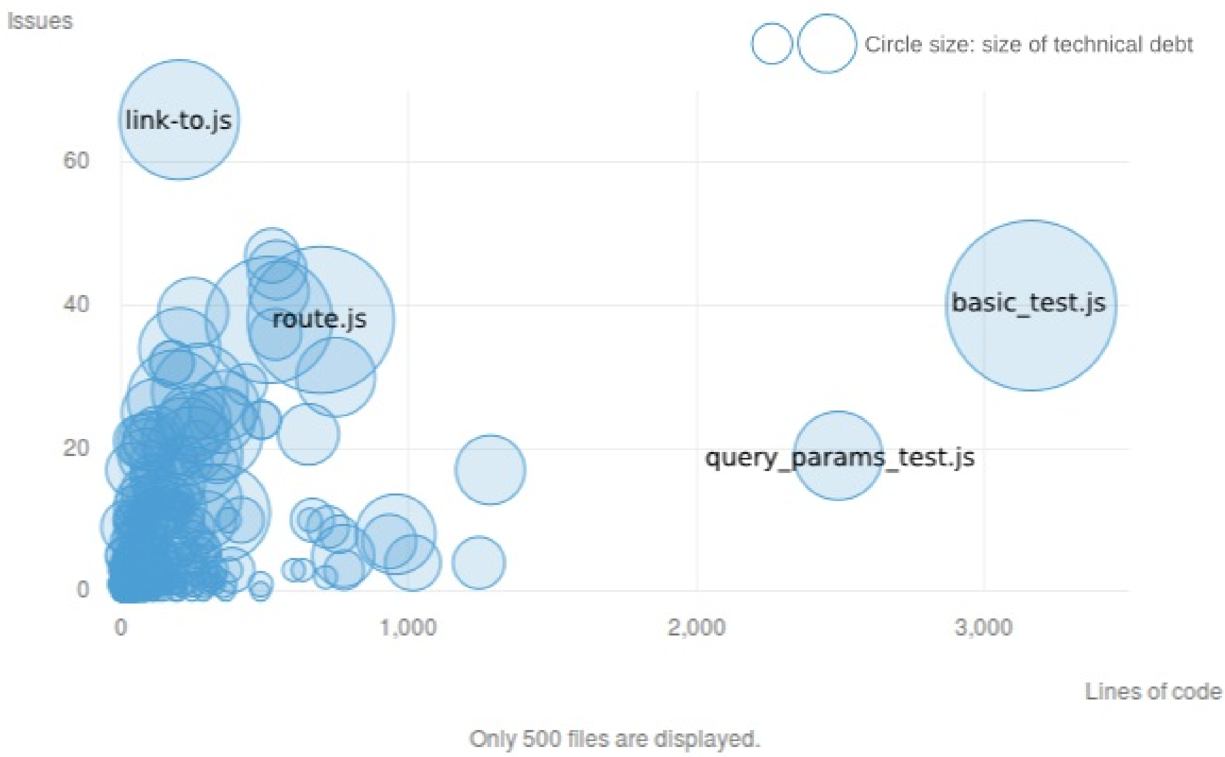


Figure 13: Technical debt on files

The `link-to.js` file is located inside *ember-routing-views* package. It has the highest issues, moderate technical debt, yet has few lines of code whereas `basic_test.js`, which is part of *ember* route testing package, has the highest technical debt and lines codes, but has a moderate number of issues. This could be a good opportunity for a new contributor to collaborate to resolve routing issues.

Ember.js: Keep Improving Performance

Significant effort has been made by Ember to improve boot, rendering, and build performance. In addition to creating a performance subteam, the infrastructure work is starting at a pace to allow Svelte Builds—builds of Ember that have all code related to deprecated features removed. This allows developers who have eliminated the deprecation warnings in their code to benefit from a significantly reduced file size.

The basic advantage of Ember is that it has slowly become a stable core on top of which developers can build web applications. Much effort is put into the *“Stability without Stagnation"* concept behind Ember and the Ember community demonstrates that, unlike other open source projects, Ember is mostly developed by the consumers instead of a giant corporation. So, the developmental stability remains without halt due to contribution from multiple users.

While low-level API gives developers maximum flexibility, it does not always make it pleasant to build large apps. However, these building blocks have unlocked frameworks and libraries like jQuery, Ember, Angular, and React. By focusing first on raw capabilities before ease-of-use, the scope is constrained. It allows Ember developers to ship features faster, and by exposing just the building blocks, the experimentation in the ecosystem can continue with the help of developers without hampering backward compatibility. The same thing is starting to happen in Ember. Two illustrative examples are *FastBoot* and *Engines*.

Conclusion: Ending the Expedition

The end of the expedition has led to the conclusion that the Ember.js framework has an interesting, but complex architecture. We believe that our chapter can help the reader to contribute to the Ember.js framework by structuring the (architectural) complexity, showing its functionalities and deployment and reveal its potential weaknesses.

Starting with the involved stakeholders, we clearly showed that there are two important stakeholder groups within the framework, namely **users** - the developers that use and support the framework to build their own web applications - and **developers**, those who continuously improve the framework. For contributions, within the last mentioned group Robert Jackson (@rwjblue), Stefan Penner (@stefanpenner), Tom Dale (@tomdale) and Kris Selden (@krisselden) are the most important to contact in Ember's Core Team. By developing a broader view, the frameworks relationships, dependencies and interactions with other systems and parties appeared. Ember.js benefit from numerous of third party software to support their framework, compatibility, community and developers. Furthermore, it is shown that as a framework, Ember.js makes use of the Model-View-Controller (MVC) design pattern, which mainly relies upon *models*, *controllers*, *views*, *templates* and *routes*. We have also shown that Ember's modules are clearly defined, but have extremely complex interactions with each other. Moreover, a generic insight in the main features of the framework along with the perspective of the stakeholders is presented. It appears that Ember.js has dozens of features, mainly for their users. Finally, we have shown the way Ember.js is deployment, its evolution, why the framework changes and how it manages the changes over time. It comes out that Ember changes because of adjustments in both *supporting languages* and *supported web browsers*.

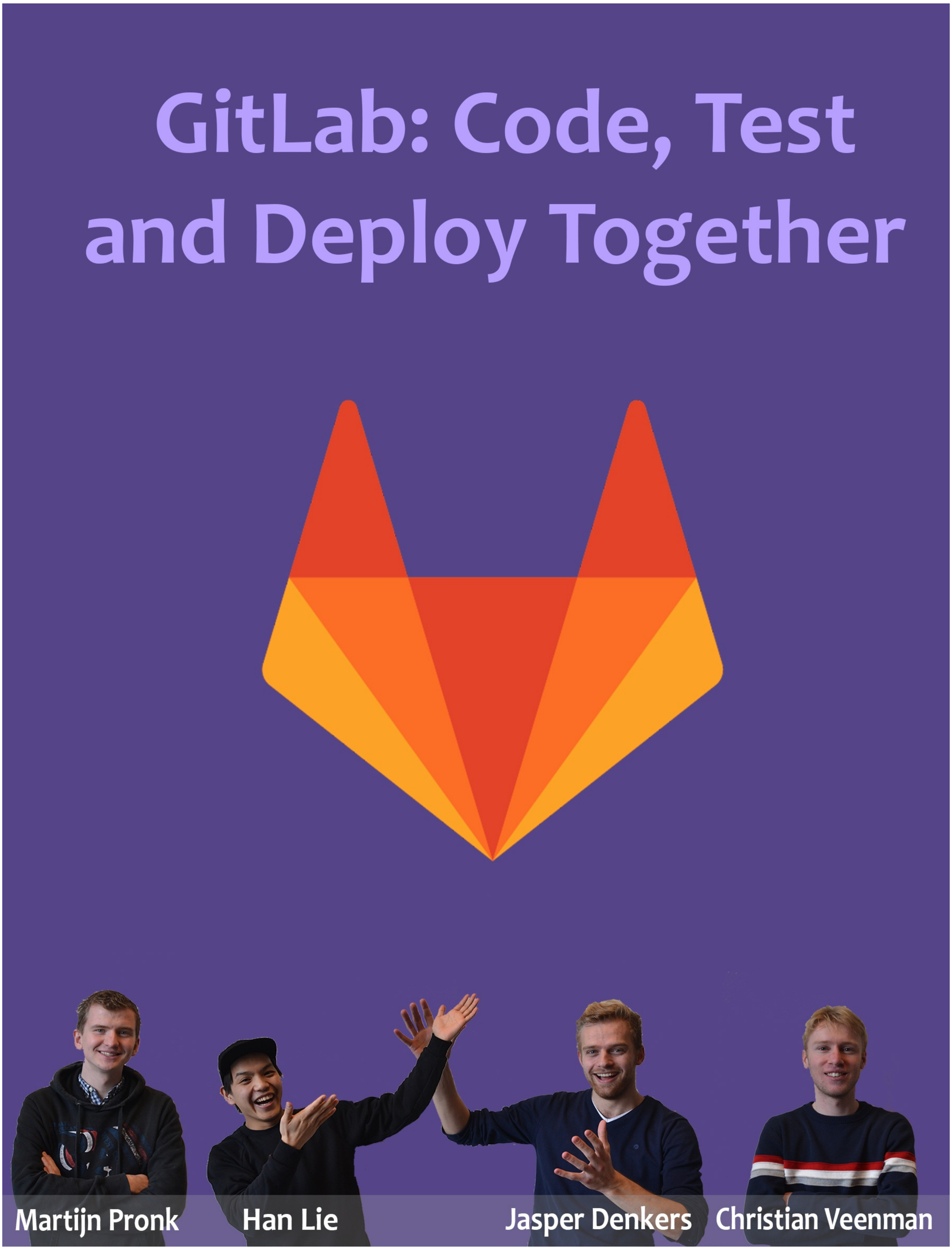
After analysing Ember.js from different viewpoints and perspectives, it can be clearly seen that the framework has improved over time. Though, there are still points of concern left. The dependencies within the modular organization make it hard to change specific components because it can be difficult to predict its consequences for other modules. This clearly has a negative impact on both scalability and maintainability of the Ember.js framework. In addition, the technical debt analysis showed that fortunately no blocking issues were found.

However, there still exists several critical and major issues, mostly about unused variables and functions. Another fact is that relatively most of the issues come from the code that handles routing and testing. To conclude, we believe that in both `link-to.js` , `route.js` (both handle routing) and `basic_test.js` there is still major room for improvement and contributions.

Though, from the intensive study, we are convinced that collaborating online on a large framework like Ember.js really moves the framework forward. Ember.js has hundreds of passionate and, above all, very intelligent contributors and warmly welcomes new contributors and developers. GitHub contributes to this as well, by supporting communication, cooperation and a structured way of working. To conclude, if you are going to build a web application using the Ember.js framework, we hope that we have encouraged you to make contributions. It is worth trying, however, small it may be.

References

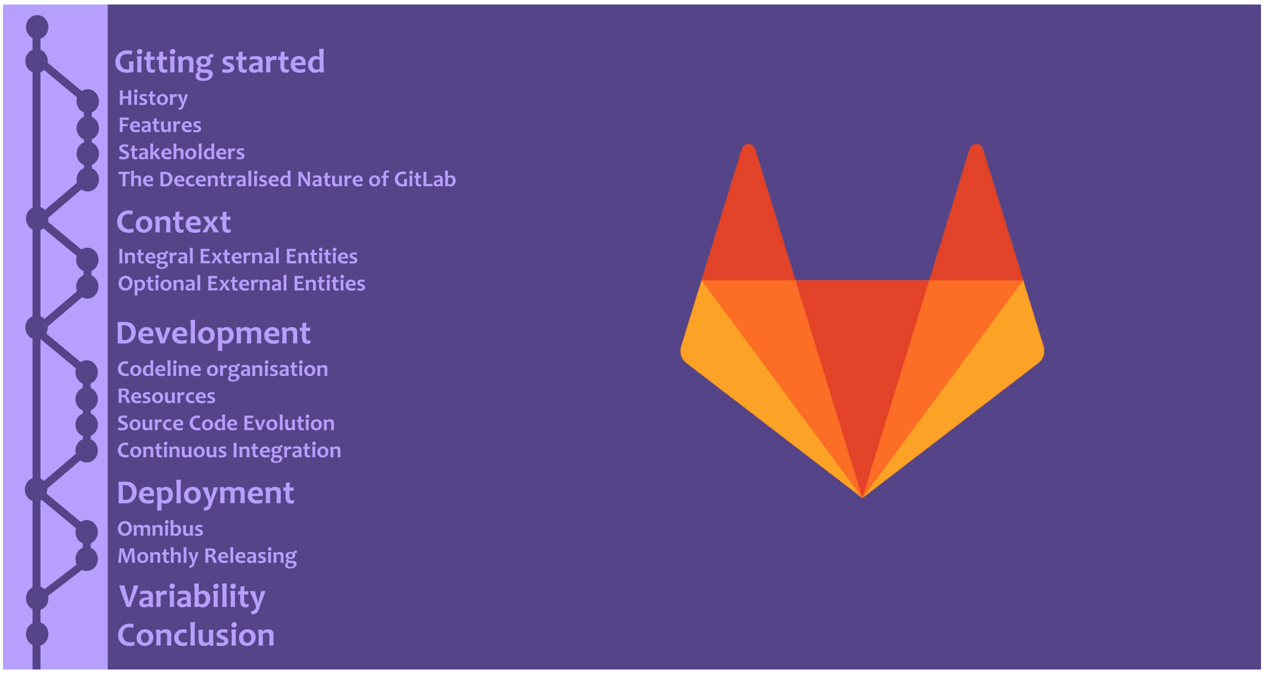
1. Ember.js Guides and Tutorials. (2016). Retrieved from: https://guides.emberjs.com/
2. Rouse, M. (March, 2011). *MVC Design Pattern*. Retrieved from: http://whatis.techtarget.com/definition/model-view-controller-MVC
3. Peyrott, S. (2015). *React Virtual DOM vs Incremental DOM vs Ember's Glimmer*. Retrieved from: https://auth0.com/blog/2015/11/20/face-off-virtual-dom-vs-incremental-dom-vs-glimmer/
4. Rozanski, N., & Woods, E. (2012). *Software systems architecture: working with stakeholders using viewpoints and perspectives*. Addison-Wesley.
5. Node.js Wikipedia Page. (March 24, 2016). Retrieved from: https://en.wikipedia.org/wiki/Node.js
6. Ember.js Blog. (2016). Retrieved from: http://emberjs.com/blog/
7. Puhlmann, F., Schnieders, A., Weiland, J. & Weske, M. (2005). *Variability Mechanisms for Process Models: Process Family Engineering in Service-Oriented Applications (PESOA)*.
8. Cunningham, W. (January 22, 2011). *Ward Explains Debt Metaphor*. Retrieved from: http://c2.com/cgi/wiki?WardExplainsDebtMetaphor
9. Penner, S. (April 21, 2014). *EmberConf 2014 Software Productivity and Ember: Ember CLI by Stef Penner*. Retrieved from: https://www.airpair.com/ember.js/expert-stefan-penner
10. Campbell, A. (May 7, 2015). *SonarQube Issues*. Retrieved from: http://docs.sonarqube.org/display/SONAR/Issues



GitLab: Code, Test & Deploy Together

Abstract

GitLab is a web-based source control manager. It focuses on documenting, managing and enhancing the workflow of software projects. GitLab distinguishes itself from similar systems by providing out of the box continuous integration and self-deployment. The system is created with Ruby on Rails, PostgreSQL and Git to implement an MVC-based architecture. It delegates tasks to specialised services such as Redis and Nginx, for performance optimisation. Deployment and distribution are done with the GitLab Omnibus package, which makes setting up a GitLab instance easy. GitLab's open and decentralised business philosophy (its community edition is open-source), results in the need for good documentation, communication and a well-structured architecture.



Gitting started
History
Features
Stakeholders
The Decentralised Nature of GitLab
Context
Integral External Entities
Optional External Entities
Development
Codeline organisation
Resources
Source Code Evolution
Continuous Integration
Deployment
Omnibus
Monthly Releasing
Variability
Conclusion

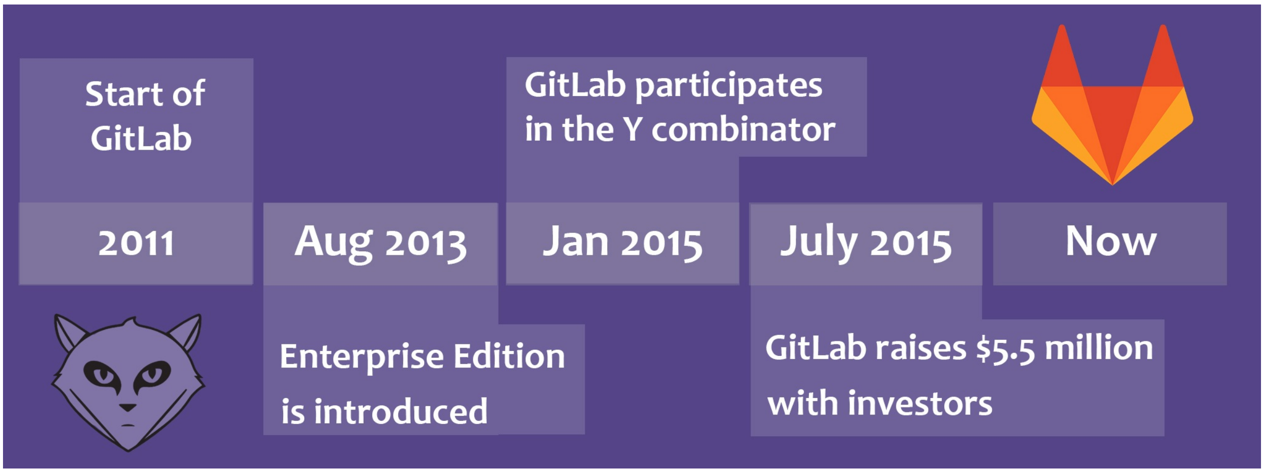
Gitting Started

In the last decade software projects have become increasingly more complex as a reaction to the high demand of functionality. This resulted in larger teams having to maintain and improve the software. In order for a software project to succeed, it is essential to have good version control and a clear process on how to contribute to the software without breaking it. GitLab provides a web-based solution to source control with Git and aims to solve these problems by allowing developers to work together in a structured manner. Examples of features that GitLab provides are: a dedicated project wiki, customised continuous integration (CI) and the ability to host GitLab on your own private servers.

This chapter gives an overview of what GitLab is, the development process and its architecture. To better understand GitLab we interviewed GitLab employee Jacob Vosmaer, who has been the fourth employee for GitLab and has helped making GitLab to a success. Jacob is one of the core implementers of the GitLab Omnibus and became senior developer in 2013. He helped us to improve our story and add valuable information.

History

GitLab is a rather young company, it will celebrate its 5th anniversary this year. The amount of downloads is growing exponentially and multiple Fortune 500 companies are using GitLab for source control.



Start of GitLab

2011

Aug 2013

Enterprise Edition is introduced

Jan 2015

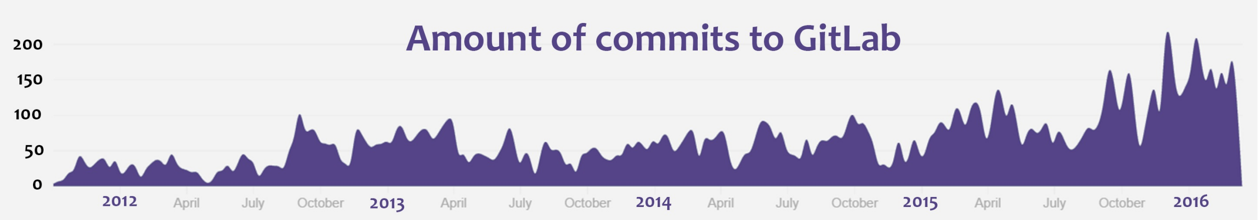
GitLab participates in the Y combinator

July 2015

GitLab raises $5.5 million with investors

Now

A brief history of GitLab can be seen in *Figure 1*. The company has been founded in 2011 and has been growing ever since. Since August 2013 a paid Enterprise Edition was made available next to the already existing Community Edition. According to Jacob Vosmaer the initial strategy of GitLab was to grow slowly. However after joining the Y Combinator (a company that provides seed money and advice for startup companies) in March 2015 they made a switch and speeded up their development.



Amount of commits to GitLab

The amount of commits is increasing by the year. The source code of GitLab is being mirrored and is available at GitHub and GitLab.

Features

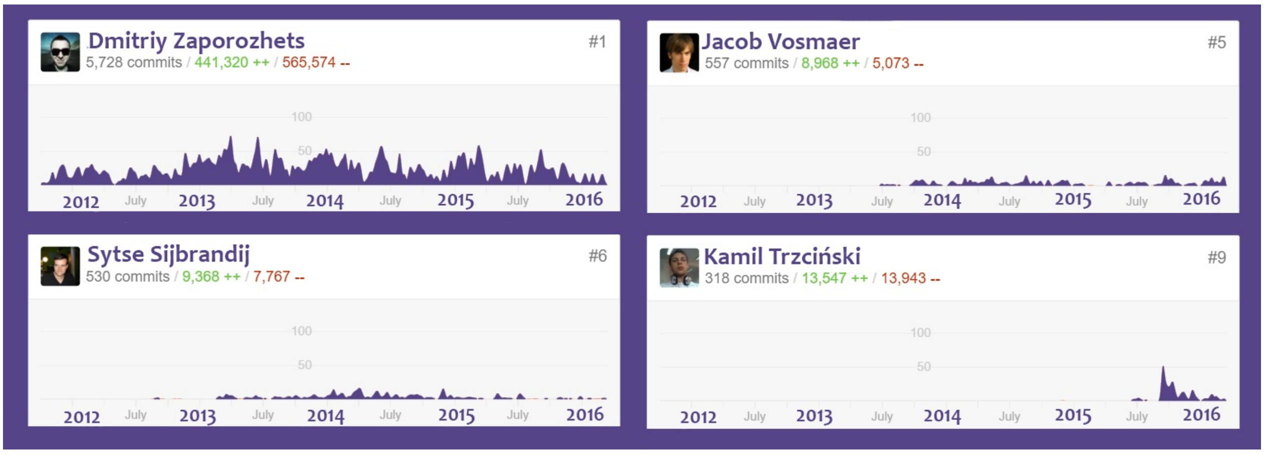
GitLab provide a lot of features that are useful when doing a software project. The features discussed in this section are thought to be the most important to GitLab.

- Sufficient **documentation** is important in every software project that multiple people collaborate on. GitLab provides a dedicated project wiki that can be accessed through the web interface. The wiki can inform users on how to build, use or learn about the software. Because the wiki is mainly edited in the web interface, GitLab provides a feature which allows users to edit files (including the wiki).
- Two main features are related to **authentication** within GitLab. It is possible to use

LDAP and two factor authentication. Two factor authentication adds a layer of security and LDAP is enabling to log in with existing external accounts.
- There are multiple **process management** features that together help with managing a GitLab project. For example adding weights to issues and the ability to view an activity stream provide useful insight of what is being done and what has to be done. Another feature that can enhance management of a GitLab project is adding an external JIRA issue tracker. When migrating between different platforms this can provide better portability.
- There are also features that together **enhance the workflow** of developers. For example, the ability to fork or mirror repositories both enhance the ability to let developers work with a repository without touching the original repository. The ability to merge builds automatically (if no errors are found) together with the feature that allows for reverting commits both enhance the experience and workflow that developers have when working with GitLab. Also the ability to do code reviews is an enhancement.
- One of GitLab's unique features is that it can be **run for free on a user's private server**, opposed to something like GitHub. The code never leaves the customers environment which is valuable for companies like, NASA, ING, CERN and Alibaba.
- There are multiple features that improve the ability to do **Continuous Integration**. The feature of integrating Docker on GitLab requires the GitLab Runner feature to be able to work. *GitLab Triggers*, *GitLab Artifacts* and *custom build scripts* all enhance the ability of doing CI.

Stakeholders

In *Figure 3*, four important people are displayed that are important to the **company GitLab**. Dmitriy Zaporozhets is the founding father of GitLab, he has made the most commits and is the biggest contributor. At the right top there is Jacob Vosmaer, who shared useful insights with us about the way GitLab works. Sytse Sijbrandij is the CEO of the company and played a vital role in bringing this idea to the market. Kamil Trzciński is the lead of CI which is a unique feature to GitLab. Kamil first started contributing and later got hired by GitLab. So from contributor he turned into an employee after doing some significant contributions. Most of the top contributors are employed by Gitlab.



As can be seen Dmitriy is contributing a lot. This made that GitLab used to have a high busfactor. A lot of knowledge was centered with Dmitriy, however throughout time the busfactor slowly decreased. The organisation attracted more employees and expertise spread through the sharing of knowledge.

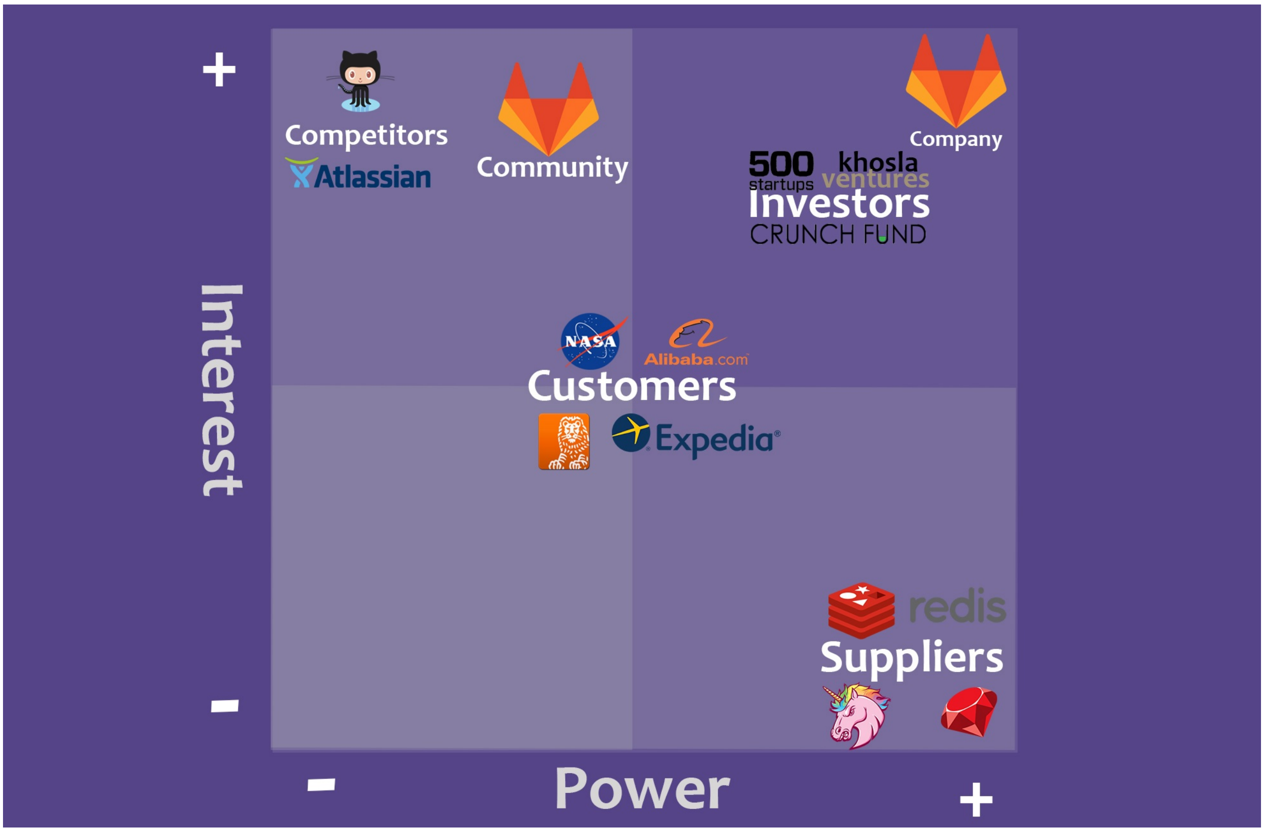
Besides the employees there is a big **GitLab community** that also contributes. A special page at GitLab is showing an overview of all the 1109 contributors (31st of March).

GitLab is not the first to offer a Git-based source-control system, there is tough **competition**. One of the most well known and used is GitHub. One of the main differences is that GitLab is a complete white box by offering the source code to the customers. This is not the case at GitHub, they are more of a black box. GitLab has gained a lot of traction in the past few years, with self deployment being one of its unique features. As has been mentioned before the GitLab repository is also shared on GitHub so the competition is helping GitLab get more exposure and growth. The competition interested in the developments at GitLab however don't have direct power to change things.

As GitLab is used to host the source of other systems, their **customers** heavily depend on it functioning properly. If GitLab does not function well, development of a system that utilises it will be hard. Things such as issue management and code review are tasks that are facilitated by GitLab and make developing less of a chore.

There are multiple **investors** that have enabled GitLab to grow further. Khosla ventures, 500 startups and Crunch Fund have supported GitLab with $5,5 million, so they have the best interest in GitLab and also have power to change things.

GitLab makes use of different open source programs. GitLab is dependent on these **suppliers** and therefore they have power since changes at the suppliers affect GitLab. From these programs there is however a low interest in GitLab. The suppliers will be discussed in further detail in the context section.



The Decentralised Nature of GitLab

GitLab is very open about their internal and external processes. Since GitLab is a decentralised organization their employees often are working in different time zones. In the interview with Jacob Vosmaer he made clear that sufficient documentation is essential since they often cannot contact their colleagues because of time differences. After their rapid exponential growth they are still able to stay open towards the community and are still working decentralised. All documentation can be found on their website.

Context

In this chapter the relationships and interactions with the system's environment (such as third party applications) are described. The context view provided in this chapter is based on the book by Rozanski and Woods.

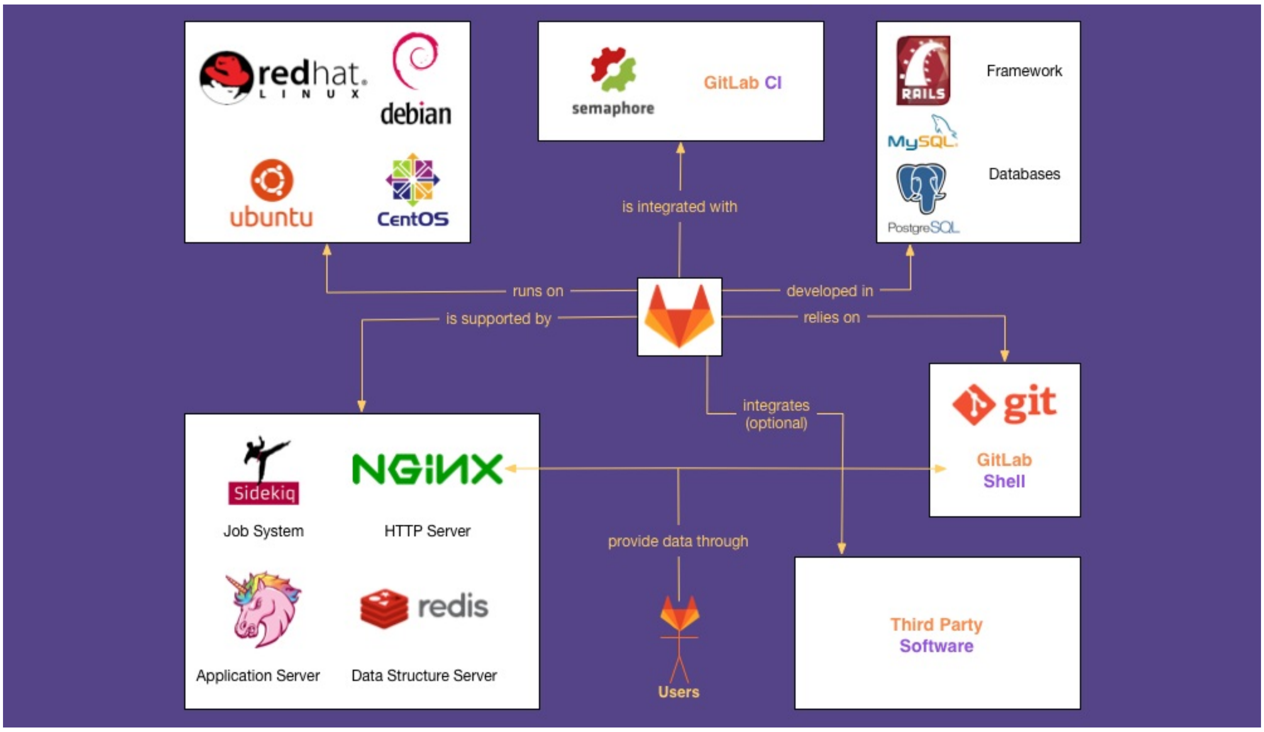
GitLab tries to achieve the aforementioned goals with an architecture that contains several external entities. Some are necessary for the system to run properly, while others are optional.

The systems architecture can be sectioned into two groups that give a clearer view on how GitLab interacts with its external entities, namely:

- Integral external entities

- Optional external entities

In *Figure 5* this overview can be seen. This figure excludes the optional external entities, as they are not a necessity for the system to run and including them would create a too chaotic view on the architecture. These will be elaborated upon below.



Integral External Entities

Development tools used to create GitLab are Git, Ruby on Rails and MySQL/PostgreSQL.

Git is both used as a development tool and as a main feature of GitLab. Git is a source control tool and facilitates users in creating a more efficient workflow. As GitLab itself is a Git-based source version control application, Git also has to be integrated in the application itself besides being used as a development tool. It is one of the most important components of GitLab.

The system is built in Ruby on Rails, which is a popular web application framework that was chosen as most of GitLab's initial developers were experienced in this framework. It is used to create applications that are based on the MVC (model-view-controller) architecture.

Databases used to store basic information are either MySQL or PostgreSQL, which stimulates portability.

GitLab also provides their own CI tool. This is a very distinguishing feature as most CI tools are paid services. GitLab makes use of this tool during development. Besides their own tool they also use Semaphore CI for some tasks.

Service providers used by GitLab mostly provide server software that try to separate the workload and make the application run more efficient. For GitLab to function properly these servers need to be installed, with frameworks and applications that talk to them. In *Figure 6* a diagram can be found that gives a better understanding of the main architecture behind GitLab and its surrounding systems. Below that figure a more thorough analysis will be conducted.

Sidekiq is a background processing framework for Ruby. It is used for job processing tasks such as queueing and job prioritisation.

Sidekiq makes use of the Redis as a database backend. It contains job information, meta data and incoming jobs. Redis is a fast data structure server, that lives in the RAM memory. This makes it great for job caching and working with Sidekiq to do the job processing.

GitLab repositories are accessible via two front end systems: GitLab-shell and Nginx.

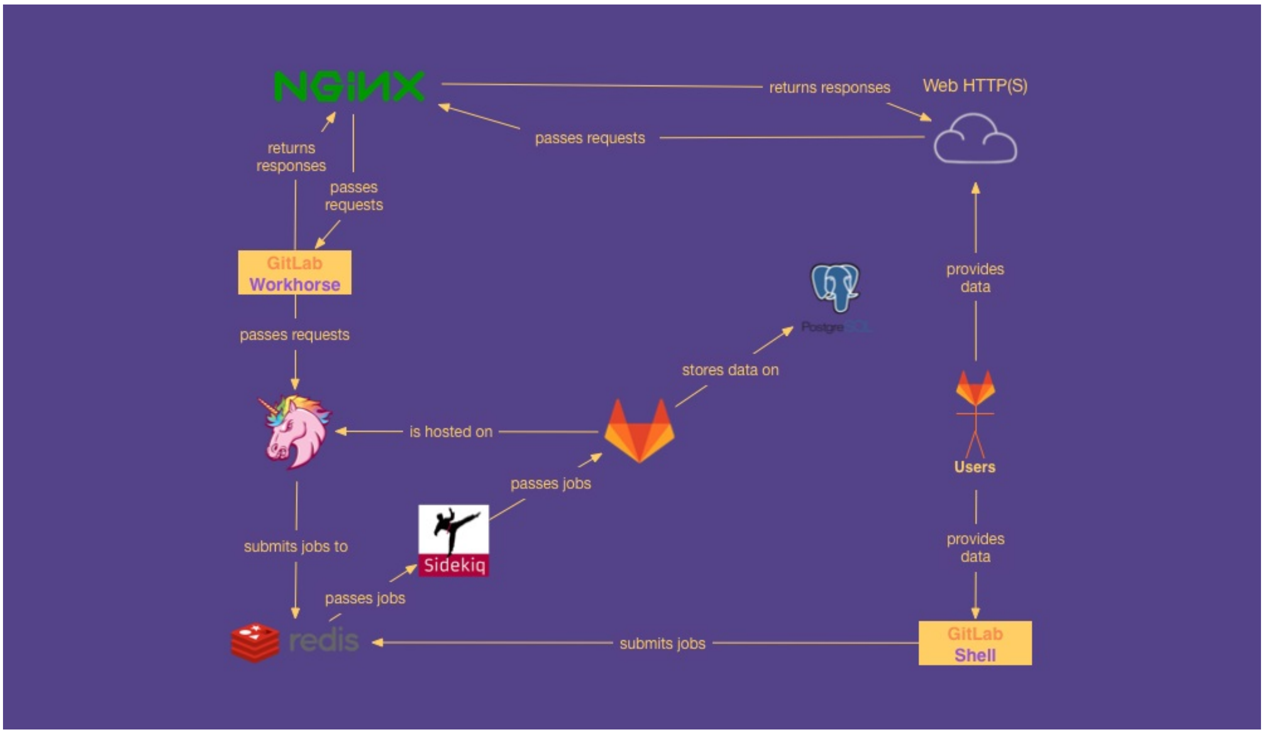
The GitLab-shell is an add-on provided by GitLab that gives a terminal like environment to handle Git commands. It communicates with Sidekiq via Redis to give Git commands.

Regarding the latter, Nginx is a web server that can handle a high number of connections and specialises in load balancing. It can be accessed via HTTP(S) and can serve static files very well. Nginx functions as a first entry point to all web requests and is used to access the Unicorn application server.

Unicorn is an application server that contains the main Rails application. It processes requests received and filtered through Nginx and GitLab Workhorse and returns a response to the users.

GitLab Workhorse stands between Unicorn and Nginx. It is used to handle large HTTP requests such as file transfers and Git push and pull requests. Other requests are passed to Unicorn to handle.

GitLab can be run on multiple server distributions. These include Red Hat Linux, Debian, Ubuntu and CentOS.



Optional External Entities

The external entities described in this section are optional to include in the system. GitLab is unique in comparison to its competitors in the sense that it has support for external entities out of the box (the integration is implemented by GitLab itself). A competitor such as GitHub can integrate these entities too, but the integration initiative needs to be taken by the companies that provide these entities themselves. Below a list is given on these entities and what they are used for.

Continuous integration: even though GitLab provides its own CI tool, it is possible to use external CI systems, including:

- Atlassian Bamboo CI
- Buildkite
- Drone CI
- JetBrains TeamCity CI

Project management and issue tracking: it is possible to use tools to enhance project management. Codeline managing, bug tracking, issue tracking, task tracking etc. can be done by integrating the following tools:

- Assembla
- PivotalTracker
- RedMine
- JIRA

Deployment: deployment can be facilitated by using e.g. one of the services listed below:

- Docker
- Heroku

GitLab facilitates integrating a lot of services. The ones mentioned above were deemed most important, as they make certain cumbersome tasks a lot easier when integrated with GitLab. Other services include spam protection, dependency/vulnerability tracking, authentication, communication, metrics and avatar integration. On the GitLab website a full list can be found.

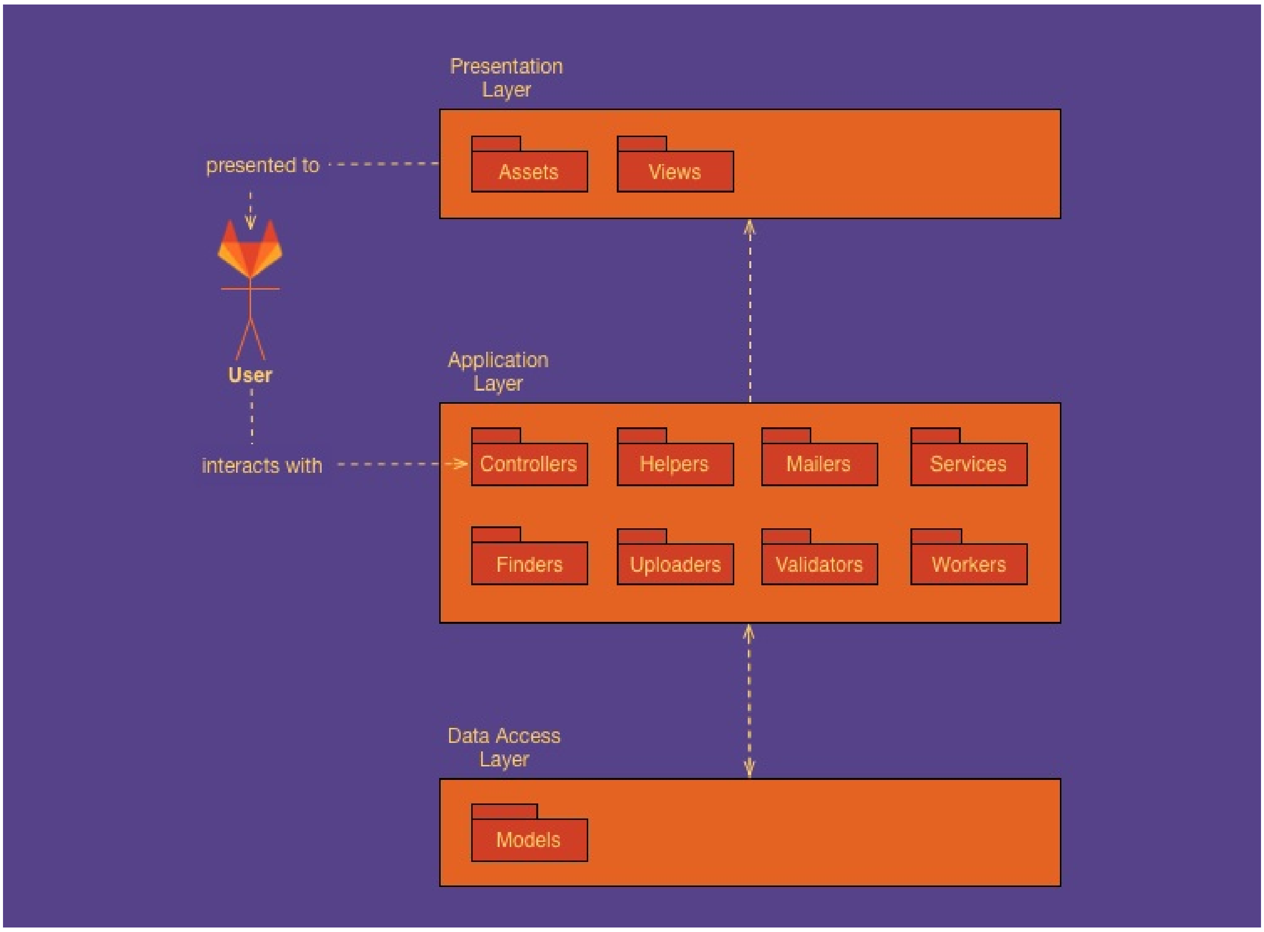
Development

This development view presents an analysis of how GitLab is structured, how the code is organised and it describes the most important processes and resources related to developing and testing GitLab.

Codeline Organisation

GitLab mainly uses the Model-View-Controller (MVC) architecture. MVC is a design pattern often used in software architecture. It separates the system into three interconnected parts, namely a model, view and controller. The *model* represents the business domain, where data can be stored and retrieved. The *view* is used for presentation to the user. The *controller* is the communication link between the model and the view. It manipulates the model by sending commands. Commands can also be sent to the view so that the right information is displayed.

According to the book by Rozanski and Woods, source code can be organised into modules that contain related code. Though MVC gives a clear view of the global structure of a system, it does not encompass all of the modules in GitLab. A module structure diagram is therefore created that takes MVC and all of the modules into account. In *Figure 7* a module structure model is given to show the dependencies between these modules.



MVC is implemented making use of Ruby on Rails, a web application framework that purely focuses on this design pattern. Therefore most GitLab modules can be separated according to how Ruby on Rails modules are organised. We separate modules into three layers:

- Presentation Layer: Contains modules that create the view of the system. Views in the presentation layer are shown to a user through a browser or console. Furthermore assets that are needed for the view are contained in the presentation layer, i.e. images, stylesheets, javascript files for interaction and fonts.
- Application Layer: Contains modules that contain the main functionalities of the system. It furthermore directly speaks to the data access layer to extract, write to and manipulate data. Following an overview of each of these modules:
    - Controllers: Process requests to show certain views to the user or to access the correct data from the data access layer. It directly receives its requests from the user.
    - Helpers: Used to support controllers. By default in Ruby each controller automatically includes all helpers. GitLab has around fifty helper modules, which focus on supporting the controller modules through for example a SearchHelper module that contains autocomplete methods for searches.
        - Mailers: Allows sending emails from the system. An example is the Groups module that focuses on sending and managing group emails.
    - Services: Contain basic functionalities of GitLab such as the MergeRequests

module that executes merges.
    - Finders: Mostly used for sorting and finding modules.
    - Uploaders: Used for managing files in directories.
    - Validators: Contains modules that validate certain properties of the system, such as an url_validator module which validates each uploaded url.
    - Workers: Contain background workers that are needed for background processing after certain actions are performed. Most of the worker modules include a Sidekiq module for job queueing their background processes.
- Data Access Layer: Contains implementation of the domain model.

Resources

Since GitLab has so many external dependencies, getting started with developing for GitLab could be hard. GitLab facilitates developing by providing the GitLab Development Kit (GDK) for various operating systems. The GDK provides an isolated environment for developers that is easy to setup up and eliminates the hassle of setting up a lot of services and dependencies to get started with GitLab development. Additionally, it accompanies the process of updating between different versions, including for example database migrations. It is unique in the sense that it encourages the use of Ruby and dependencies installed natively on your OS, rather than in a virtualised environment. Due to lots of IO operations, a virtualised solution would be much slower for running and testing the system.

Since GitLab is a continuously evolving system, no specific time constraints are set. Additionally, it seems like the organisation does not have specific budget constraints. There is budget, as provided by the investors and customers, which is used the pay employees and evolve the system as quickly as possible. The only self-imposed time constraints that exist are prescribed by the monthly release schedule.

The main resources that are used to develop the system are developers. There are two types of developers that build the system: GitLab employees and open source contributors. The GitLab employees are mainly remote workers and get paid. The open source contributors provide free contributions where GitLab benefits from.

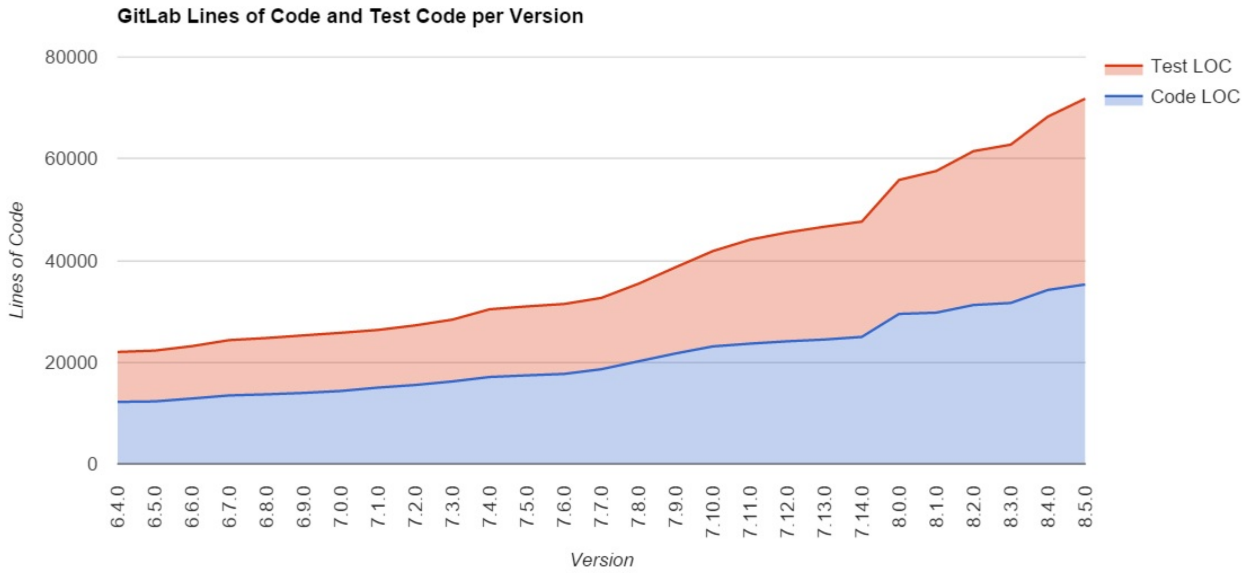
GitLab provides a free distributed Community Edition of their system, available for dedicated installation for developers the use for their own projects. This distribution requires hardware in the form of servers. When a consumer decides to run an instance of the GitLab community edition itself, hardware is required. Additionally, hardware is required for the so called GitLab runners, the workers that execute CI builds. GitLab also provides free CI runners. In practice, these are slow and real customers would probably decide to add extra resources by dedicating servers for building their GitLab CI builds.

Probably the most important form of development resources is the documentation. GitLab focuses on openness and transparency in communication. This is reflected in the amount of effort that is put in online discussions, guides and management of issues and merge requests. This empowers employees and contributors to quickly get started with contributing. Additionally, extensive feedback on proposed changes helps keeping quality high. The descriptions and guidelines of internal processes are also open.

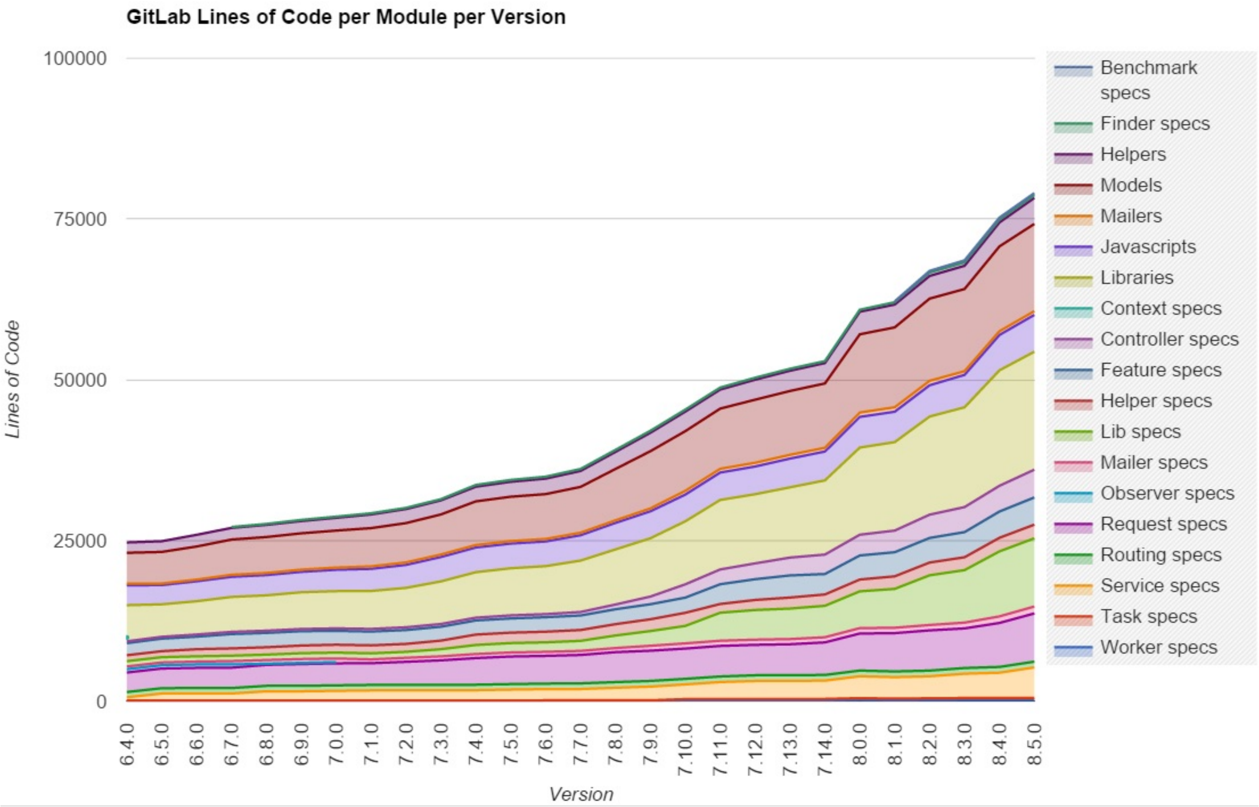
GitLab is mainly written in Ruby and its current version uses about 173 external gems. Gems are basically libraries that are written in Ruby and can be installed by the gem command which will automatically download them from a source. Reusing code is often a good idea since it saves development time and costs, often also providing a more specialised solution. However since the external gem files are hosted on the rubygems.org, GitLab is dependent on the behavior of rubygems.org. If the owner of a gem decides to remove a gem from rubygems.org that GitLab uses, it will create issues when building GitLab. A good alternative would be if GitLab mirrors all gem files on its own servers so that they are not dependent on rubygems.org.

Source Code Evolution

We measured the volume of the GitLab codebase over time. The measurement is performed on major/minor versions (x.x.0). The results are visualised and analysed below. The graphs in the proceeding sections visualises the evolution of GitLab’s codebase over time.



In *Figure 8* we see that the overall (normal and test) code volume increases gradually. For version 8.5.0, it increased with a factor more than 3 relative to 6.4.0. Additionally, we see that the amount of test code relative to normal code increases over time. For version 6.4.0, the amount of normal code is 1.24 times more than the test code. At version 8.5.0 it is less: 0.97. This could indicate that GitLab's focus on testing is increasing.



With the visualisation in *Figure 9* we see that the growth of the overall codebase is spread across most modules. There are no single modules that increase substantially more than others. Most modules are present from the beginning till the current lifecycle and no splitting of modules can be identified. This could be a sign that good architectural decisions were made at an early stage, since there was no need for big refactoring afterwards.

Continuous Integration

GitLab performs CI using two systems, namely Semaphore and their self created GitLab CI. Via these systems it is made easy to do automatic software testing and deployment to the cloud. Extensive automated testing increases quality during development. As we have seen in the Source Code Evolution Analysis, this is of big importance for GitLab. This is supported by one of the contribution acceptance criteria, which states that all contributions should contain proper tests.

GitLab builds itself using their own tool. This is a nice demonstration of one of their own prominent features, CI within the GitLab environment. CI builds are parallelised and sped up by dividing the complete build job into several dependant and independant smaller jobs. If a job fails, jobs depending on such a job will not get executed to save resources.

Additionally, CI is applied using Semaphore for targeting tests with specific setups that GitLab does not support itself, yet. This mainly consists of tests using PostgreSQL as a database.

Deployment

Interestingly enough, since GitLab is a piece of software itself, it is used as the main tool for coding, testing and deploying itself. In this section we will focus on the deployment part.

The GitLab Omnibus

GitLab deploys and distributes its system to consumers in a package that's called the GitLab Omnibus, which was one of the big factors that allowed GitLab to grow. Since GitLab relies on a variety of external services like databases, servers, etc., setting the system up could be a hassle. The Omnibus package solves this by packaging required configuration options and updating and maintaining features into a single resource that users can install on various operating systems to get started quickly in minimal time. Initially, GitLab was mainly supporting Ubuntu 12, which was the most recent stable Ubuntu release back then. However, because larger companies often used Red Hat as their Linux distribution, it often caused minor issues when following the manual installation and configuration steps.

Jacob Vosmaer played a significant role in the creation of the GitLab Omnibus. In our interview with him he explained to us that creating the GitLab Omnibus decreased the time necessary to invest in technical support since most users did not have to do any manual configuration steps anymore. The user could simply install the GitLab Omnibus package. Upgrading your current GitLab configuration can also be done by using single-set Omnibus operations, which removes the need for manual configurations when a new version has been released.

The GitLab Omnibus is largely based on CHEF, because CHEF is easy to configure for multiple platforms. Currently the GitLab Omnibus contains Nginx, LDAP, PostgreSQL, Redis, Unicorn and more systems required for running.

Monthly Releasing

Since October 2011, GitLab has consequently released a new version on the 22nd of each month (excluding patches and security releases). It is the 22nd of the month because just the last release before GitLab decided to start releasing monthly was on the 22nd, and they decided to stick with that.

The philosophy of GitLab prescribes that there should be no single point of failures. We can see this in the release process, where there is the role of a *Release Manager*. This is a role that is being passed on to a different person every month. The two main takeaways of this method in the deployment is improved reliability (the busfactor is decreased) and a better process. It gets a refreshing look every month and documentation is stimulated. This enabled GitLab to release monthly without exceptions, while working with mostly remote developers that do not have frequent physical meetings.

Variability

GitLab is built to be very configurable. Projects have different demands and therefore also have a need for different features. The variability points that are of main importance to GitLab will be discussed in this section.

- **Community vs. Enterprise Edition:** The product is distributed in two different ways: a free Community Edition (CE) and a paid Enterprise Edition (EE). Selling the EE generates the main source of income for GitLab. It varies from the CE in the sense that it has extra features that are developed by GitLab and is mainly targeted at bigger organisations. Those features include advanced authentication and authorization options, extended workflow and project management, etc. However all contributions made by the open source community are always added to CE and will never be made only available for the EE. Another difference between the CE and EE is that EE includes customer support whereas CE does not. GitLab itself maintains an extensive list on their website viewing the differences between CE and EE.
- **Authentication and Authorization:** While GitLab contains a basic authentication system, the authentication system can be configured to a project's needs. GitLab can be configured to make use of multiple LDAP servers, two way authentication or Kerberos user authentication. Two factor authentication can be used together with other forms of authentication.
- **Project Settings:** GitLab provides various project supporting features out-of-the-box which are automatically enabled (like issues, continuous integration), but some users possibly are not interested in using those features at all. Therefore, these features can be disabled on a per-project basis. This is being done by either disabling the feature in the web interface or by adding the configuration options in the GitLab.rb configuration file which can be found in every GitLab setup.
- **Operating System (OS):** GitLab deploys its system either via the Omnibus package or by doing a manual installation. By using the Omnibus GitLab was able to provide more support for different platforms. Whereas at the start GitLab was only available on Ubuntu, it can now run on a variety of platforms including CentOS and Red Hat Linux.

Conclusions

The main goal of GitLab is to code, test and deploy together. It tries to achieve this by providing a web application that integrates Git repository management, code review, issue tracking, project reporting, continuous integration and deployment.

GitLab distinguishes itself from similar applications such as GitHub in the sense that it makes self deployment possible and easy. It delivers an out-of-the-box experience with its GitLab Omnibus and includes CI to make development and testing even more accessible.

GitLab has a culture and philosophy that embraces openness and transparency, which is necessary as they are a company that functions mostly decentralised. This results in the need for proper documentation, good communication and a strong architecture.

The main architecture created using Ruby on Rails makes GitLab a well organised system, that facilitates integrating external entities such as Redis and Nginx. Integrating them makes the system faster by dividing time consuming tasks over specialised services.

Currently GitLab is getting a lot of traction. They have received several large investments in the past few years and GitLab is becoming more popular due to their free features of CI and self deployment. In upcoming years it hopes to grow more and invest more in these distinguishing features.

As indicated by Jacob Vosmaer, the focus in the near future will be on extending CI functionalities. In the past few releases, new technologies like the programming language Go are introduced. These two moves and their impact on the evolution of GitLab will be interesting to follow.

Guava - As it Currently is

Bastiaan Reijm, Marco di Biase, Qianqian Zhu, Luca Pascarella

Delft University of Technology

Abstract

Guava is a Collections and Utility library, focusing on complementing the Java Standard library. We present the current state of Guava for newcomers and interested parties. We look at what Guava is and elucidate the architecture by analysing several key viewpoints and perspectives. Furthermore we present the current plans for Guava so that contributors and users know in which direction the project is going. We briefly discuss what to do when contributing. Finally we look at our own work and experiences at we gathered while analysing this project. Our aim is to provide an outside perspective of Guava.

Table of Contents

Introduction

The Guava project [1] is one of the many libraries written to expand and simplify the use of the Java Standard Library (JSL). This chapter about Guava is an academic study conducted to analyse this project and show it from multiple angles. What is it? Who is involved? Why use it? How to contribute? All these are valid questions and this chapter attempts to answer these questions and provide resources for further investigation on that project.

First off, who this chapter written is for. Often it is hard for an outsider to get insight on the working of a tool or how a project is organised; it requires a substantial amount of effort. The Guava wiki [2] is an excellent place to start however its main focus is usage and the features that Guava has. We take a different approach. We started as outsiders to this project and now that we know more about this project we present our findings so that anyone in the near future can gain a similar understanding of Guava without expending a similar amount of effort.

With the previously elaborated idea in mind, the chapter is organised as follows. First we start by looking at what Guava actually is from both a conceptual as well as an architectural point of view. In these sections we explore in details: the stakeholders, the context viewpoint, the information viewpoint and the evolutionary and security perspective. The previous analysis are also the starting point to propose a little comparison with similar projects like Apache Commons Collections. Then we discuss the future plans of Guava and some phenomenon related to its real-world usage. Finally we conclude this chapter by reflecting on our own work and the lessons we learned.

What is Guava?

In the simplest terms, Guava is a Java library. Specifically it is a Collections and Utilities library that adds functionality to the Java Standard Library (JSL). It is not a separate language and it does not attempt to be a replacement of the JSL. Rather it provides features and functionality that are either non-existent or cumbersome in Java.

Where is it used?

As with any product, the proof is in the usage. Guava is used in several scenarios of everyday Java programming. When dealing with common Java development, enhanced functionalities on top of JSL are needed. For instance, dealing with `null` values is a hassle. Guava provides a way to handle this. Giving it a name, increases readability in the first place and forces developers to think about the cases when a method might return a null value.

`Optional` [3] replaces a nullable reference with a non-null value. An Optional may be *present* or *absent*.

```
Optional<Integer> possible = Optional.of(5);
possible.isPresent(); // returns true
possible.get(); // returns 5
```

Returning Optional makes it impossible for callers to forget that case, since they have to unwrap the object themselves for their code to compile.

Another key example of additional functionality is for Strings. Strings in Java are used in a multitude of scenarios. Guava provides enhanced functionalities in order to join, split, match characters and use specific charsets. For instance, using Guava `Splitter.split` [4] allows to control over the confusing behaviour of `String.split` when dealing with trailing separators or empty. This is just the beginning, because string splitters is much more powerful. For instance:

```
MapSplitter splitter = Splitter.on(" ").withKeyValueSeparator(":");
splitter.split("a:1 b:2"); // => Map {a=1, b=2}
```

creates a Map starting from a string, defining a custom splitter and key-value separator.

Java collections are very useful, but Guava expands classic Maps and Sets with `Multiset`, `Multimap`, `BiMap` and `Table`. These have specific features that are useful in a multitude of scenarios. For example, JSL `Map<K, Set<V>>` is the typical way to have an non-labeled directed graph. Accessing it though, is not trivial. Guava's Multimap (and in particular for this scenario a `SetMultimap` [5]) makes it easier to map one key to multiple values. A Multimap is a general way to associate keys with arbitrarily many values.

Finally, Guava `Iterable` [6] is another useful utility class that provides utility methods that operate on or return objects of type Iterable. All of the iterables produced in this class are lazy, which means that their iterators only advance the backing iteration when absolutely necessary. This is extremely useful when a *collection* is not actually stored in main memory (for instance retrieved from a database) and does not support operations like size without actually grabbing all of the elements.

These are just a few of the large number of functionalities made available with Guava.

The Architecture of Guava

This book is about Software Architectures of Open Source Systems thus we find it appropriate to include some factors, viewpoints, and perspectives that relate to the architecture of Guava.

Stakeholders

The very first thing of understanding the Guava's architecture is to identify and understand the role of stakeholders. Since stakeholders, who have interests in the system, play a vital role in the realisation of the system. They explicitly or implicitly determine the main features of the system to satisfy their needs.

To start with, a brief background information about Guava. In 2007, Guava started out as the "Google Collections Library". In 2010, Guava released its first version on GitHub. For now, Guava has 6,734 usages according to the statistic is shown in Maven Repository website [7]. Meanwhile, Guava has 1,002 copy repositories (forks) by other GitHub users [8]. In one sentence, Guava is an open source project while it is also a collection of APIs referenced throughout the internal Google codebase. The two keywords, *Google* and *open source*, determines the distribution of Guava's primary stakeholders.

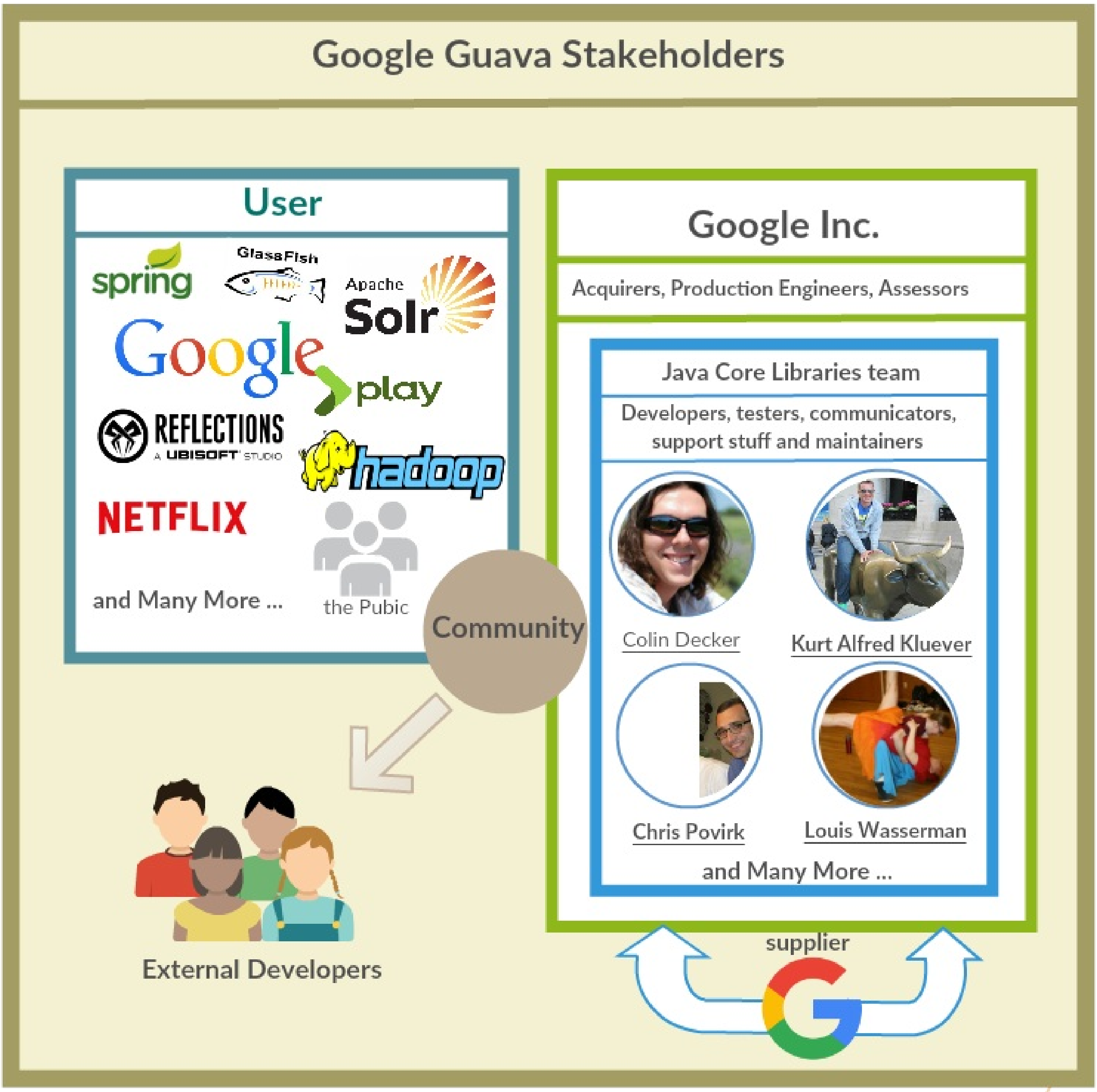
First of all, Google, as the owner of Guava, develops, tests, maintains, supports and uses Guava project. Thus, in this case, staff from Google consists of the central part of the stakeholders. Google, of course, is one of the biggest **users** and **suppliers** of Guava. **Acquirers** are mainly senior managers from Google. **Production Engineers** are members from Google that provide development and test environment for Guava. **Assessors** are from Google's quality control or conformance departments.

Among the **developers**, **testers**, **communicators**, **support stuff** and **maintainers** are Java Core Libraries team members at Google [9]. They communicate with and provide support to users in user community while they develop, test, and maintain Guava project. The most important **developers** that have contributed to guava are @cpovirk, @kluever, @cgdecker and @lowasser. Each one of them has more than 150 commits on the repository, and more than 40k of lines of code written.

Meanwhile, *open source* means the public has access to Guava and they can use it; possibly make contributions the project. In this case, third-party organisations and the members of the pubic consist of **users**. The third-party organisations include Spring, Hadoop, Reflections, GlassFish, Play, Solr [7] and Netflix [10]. The members of the public include web application developers, Java programmers, distribution application developers and CSP programmers. These users can also act as the external **developers** via GitHub. They sometimes can also fill in the role of **accessor** (pr #2247) and **communicators** (pr #2248).

To date we have not identified any **system administrators** in the Guava project, since Guava serves as a core library and no need to run this system additionally.

The stakeholders diagram below shows the overview of Guava's stakeholders as mentioned above.



Guava has a large and diverse set of stakeholders; both the community and Google actively contribute to this project.

Context Viewpoint

If we would like to summarise in one sentence what Guava wants to achieve, we could say don’t reinvent the wheel [2].

Creators and Advisors

The former Google Collections Library was born due to the necessity to extend some core Java functionalities in order to improve the structure. Two engineers at Google, Kevin Bourrillion and Jared Levy, decided to step in and write an extension to the standard JDK, which included productive features such as hashing, functional programming and so on. Apache Commons Collection was in the game, but unluckily was not enough for the kind of structures and operations that these two people were looking for.

Two big names were prominent in the original design of Guava [14]: one is Joshua Bloch, the original lead designer of the Java Collections framework. The other is Doug Lea, one of the lead designers of concurrency utilities in JDK. They both advised and reviewed this library’s design and preliminary implementation.

Guava is a Java library that lets any project to import it and make use of its features. In contrast to the JDK, Guava deprecates and eventually deletes unwanted features over time. The changes are made in order to make sure that no actual system that uses it gets broken. New components are also very important as well, and they are marked with the tag @Beta. They receive the same treatment as normal component, regarding for instance the testing aspect. If Beta component are to be deleted, they are deprecated one release before.

Internal organisation

From Guava's codebase, we found it consists of four projects: `guava` , `guava-gwt` , `guava-testlib` and `guava-tests` . We will provide further analysis of Guava internal organisation dependencies in Development Viewpoint section.

External dependencies

From Guava's maven build file, we identified the several external dependencies. Firstly, Guava depends on Java basic packages JDK and Javax. Secondly, some web server tools are also required such as Jersey, GWT. Thirdly, Guava injects dependencies with Guice[30]. Fourthly, Guava's testing related dependencies. Apache.common.math package is used to conduct benchmark testing. Junit is used to conduct the unit testing. Several Java bytecode generators, such as cglib and ASM, is also used for testing. In addition, Joda-Time, which provides a quality replacement for the Java date and time classes, is also adopted in Guava.

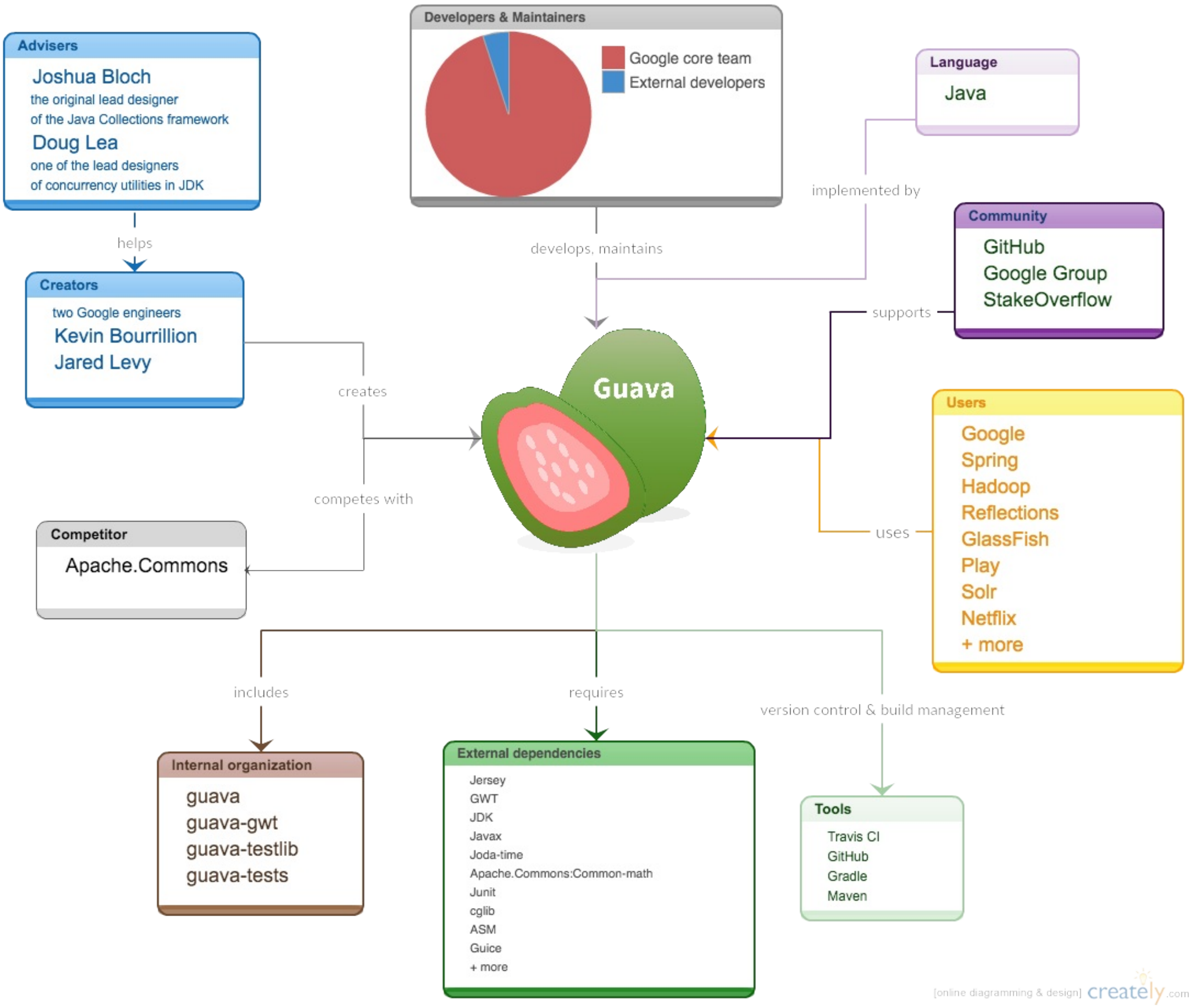
User and Community

The users can get supports and helps via User Community. The main websites are GitHub, Google Groups and StakeOverflow. Public users can post questions, feature requests or potential patches in the community. The developers are also involved in the discussions.

Version control and build management tools

Guava is a open source project and it mainly uses GitHub as its version control tools. Travis CI, as a popular companion with GitHub for open source project, is adopted to build and test projects Guava also uses Maven and Gradle for build automation.

The context model is presented below (context model diagram). The context model diagram summaries the above content and also includes information from stakeholders analysis. In conclusions, Guava's context model is relatively simple, since Guava is a Java library and does not involve much common processing.



Development Viewpoint

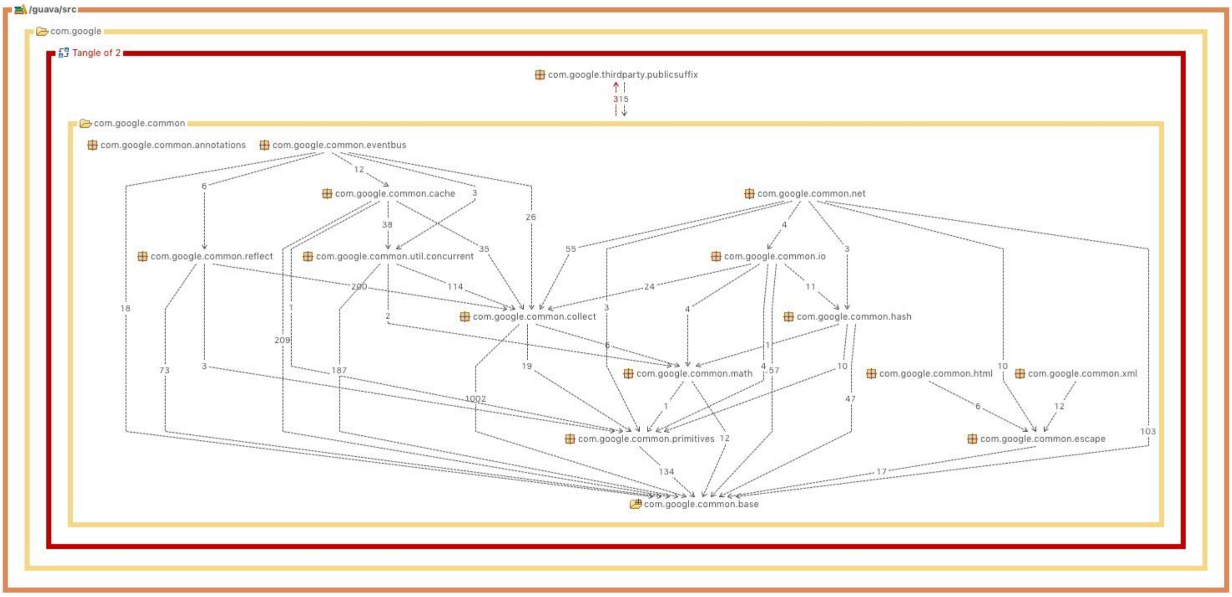
The development view gives more information about the organisation of the project, in particular regarding its source code. This view focuses on the technologies that are used in building, testing and releasing a software product.

Module Organisation

Once you set up a local clone of Guava on your machine, you will find the current version (version 19) of Guava consists of four projects:

Project	Purpose
`guava`	the main source of Guava
`guava-gwt`	the GWT-compatible super-source that helps make Guava GWT-compatible
`guava-testlib`	the test suite builders that assemble customised, exhaustive test suites for collection implementations
`guava-tests`	the tests for Guava

Since `guava` project represents the main source of Guava, we further analyse the `guava`'s module organisation. The project `guava` consists of 16 packages. Guava Dependencies Figure generated by using STAN4j Eclipse plugin displayed the inside dependencies.



Based on the package dependencies and functionalities, `guava` project can be further clustered into four layers:

Layer	Package(s)	Functionality
The corest of the core	`base`	provides with fundamental utility libraries and interfaces for other modules.
Utilities Module	`primitive` , `escape` , `math` , `html` , `xml` , `io` , `net` , `concurrent` , `collect` , `cache` , `eventbus` , `hash` , `reflect`	each functionality can refer to Guava's API documentation [12].
Annotation Module	`annotation`	served as an independent module which provides common annotation types throughout the whole project.
Public Suffix Module	`thirdparty.publicsuffix`	accessing to public-suffix information (public suffix can refer to [13]).

Standardisation of Design and Testing

The participation of two well-known Java fundamental designers, that are Joshua Bloch and Doug Lea, implies that design standards in Guava are quite elaborate and strict. Based on Guava's wiki documents as well as the previous investigations of Google Java code guidelines, we conclude that several design strategies are split into the following parts:

- best solution
- intuitive semantics of methods and classes (details refer to good code style [15])
- focus on generic use cases
- emphasis on maintainability:
  - most exposed classes should be `final`
  - conservative attitudes towards `AbstractXXX` classes
- compatibility:
  - introduction of `@Beta` annotation which indicated the related features are not ready to freeze

Furthermore, Guava also has been battle-tested in production at Google. For now, the guava-tests package includes more than 600,000 individual test cases which achieve thorough testing coverage. Guava also has additional benchmark tests (in `guava-tests/benchmark` subfolder) to check the real performance.

From the wiki and codebase, we identify that Guava has used at least three **test tools**:

- JUnit: unit testing framework
- Maven, Gradle: build tools manager for Java
- Travis CI: automatic test execution

Information Viewpoint

From an Information Viewpoint Guava has a rather simplistic architecture. Guava itself inherently does not have any information; there is neither a database nor a configuration file for Guava. Guava does not have the common ownership, mapping, volatility, and consistency problems that other systems have. Guava is not even a complete system, rather it's a utility library. Therefore in this viewpoint we focus on information purpose, usage, flow, quality, consistency in the context of what does Guava do to try to help with these aspects.

A good example of how Guava is more than just a wrapper for the JSL is the Input and Output (IO) [16]. Guava uses a paradigm commonly found in graph theory, namely sources and sinks. Information is read in by a Source and written out by a Sink. To facilitate this paradigm, abstract classes are provided that support this concept and aim to help developers write more consistent code with ease.

IO certainly is not the only part where Guava provides useful features. Among the more mature and popular parts of Guava — according to the wiki [17] — Guava provides extra functionality with regards to working with collections. For example, Guava provides functionality to take the cartesian product of two sets. Remember that the cartesian product is the set of all the pairs that can be created by picking one element from each set. This functionality is not available in the Java 8 or lower. While this is a rather trivial example, Guava provides much more functionality like this for Collection types.

Another significant area that Guava targets is data consistency, it does so in the form of caches [18]. Caches in Guava provide ways to manage data eviction strategies so that caches limit their memory footprints. These caches are most useful when memory can be traded in for processing speed, data will be queried more than once, and the entire cache can be loaded into memory. Guava caches are local in-memory data structures that neither write data nor persist over multiple applications runs.

Guava as a project is of low complexity from an Information Viewpoint simply because it does not store data, it only facilitates the processing of data. This functionality comes in the form of IO handling, collections manipulation, and in-memory data management.

Evolution Perspective

Like most software systems and libraries, Guava has changed over time. The project started as the internal collections library for Google and became known as the Google Collections API.

The major architectural changes are always discussed with the community, however seeing these changes in action is a different story. Google maintains Guava with input from the community. Google maintains an internal issue list and has internal pull requests [19]. Thus changes are added to the internal repository, tested, and mirrored out. In theory this model safeguards the reliability of the changes though we have no of measuring this effect. It also seems to increase the timescale for changes since discussions on the forum can be quite elaborate, involving many stakeholders. The discussion about how to handle Java 8 started in July 2014 and the release that is scheduled to adopt these changes will become available in early to mid 2016.

Java is a major external factor driving changes in Guava's implementation and design. Java 8 introduces duplications of functionality, incompatible features, and the need to separate Guava into several versions [20]. Not only do major languages changes to Java affect Guava but so do minor security updates. When Java 7 update 51 was released, a small security fix was introduced that directly affected Guava and had to be fixed in a timely manner. Issue #1635 elucidates the details of this issue and the fact that this only took 24 hours to fix.

Smaller updates and additions offered up by the community take much more effort to be included. Feature requests are actively discussed but seem to rarely make it into production code and if they do, it's only at major releases when the Google team is content with the stability of the system. The pull-requests tell a similar story, many of them are simply waiting to be merged [21].

Guava is managed by Google and ultimately this relationship causes changes of smaller magnitude and a seemingly longer timescale for most changes. It has much active discussion and many interesting challenges such as that a large part of the evolution of Guava is the evolution of Java.

Security Perspective

Guava, as mentioned several times before, is a utility library. It does not have it's own system state and it does not have to directly protect any sensitive assets.

Guava itself may not need many security features but maybe surprisingly does not provide security oriented features. Why is this the case? It seems that the community simply is not interested in security features. For example, according to the exchange between a community developer and a google engineer about the escaping of html characters, the google engineer states that they simply have seen any demand to justify changing the current implementation issue #1887. Less than 1% of all the issues are security related and of the 98 pull requests not one of them is security related. Similarly less than 0.5% of the messages posted to the google groups forum are security related. Due to the lack of any substantial communication related to security we conclude that the Guava community is not a security oriented one, thus Guava does not provide extra security oriented functionality.

Guava does however use one narrow facet of the security features provided by the JSL. In order to efficiently store and access data, objects are often assigned a number (a hash) in a process called hashing. Guava allows developers to more easily create hashes using cryptographically strong algorithms that specialise in uniformly distributing these hashes [22]. The advantage is that data can be retrieved faster if fewer objects are assigned the same hash. As such the security features are used because they are also advantageous in increasing the efficiency of collections.

As for the most important asset in the Guava ecosystem, the source code, it is secured against malicious code injections. In order for malicious code to be intentionally added to the code base, a pull request would have to be approved by a google team member and pass internal testing before being mirrored out the public repository. This mirroring and internalisation process means that Google can check the quality and functionality of the code before it ever reaches the general public. Another important security measure is that releases are available on the Maven Central Repository and are protected using checksums to verify the identity of the packages (see issue #354).

Guava, in our opinion, is a low risk project with respect to security. Inherently it does not have any sensitive assets other than the source code. Standard security measures to ensure the integrity of this code base are in place and the internalisation of contributed code allows for security audits before the code is made public. Since it builds on the JSL general security flaws in the JSL and JVM will also affect Guava. We also conclude that the Guava community and Guava as it currently is, do not have security as a main focus.

Comparison to Apache Commons Collections

Some of you readers may be familiar with the Apache Commons Collections (ACC), this is a similar project. However, Guava has some distinct advantages.

First and foremost Guava has had more recent activity than ACC. As of this writing 11 major versions have been released with a median release frequency of 133 days as measured in the last 5 years [23]. ACC in comparison has had 1 major release in that time period because the project was not actively maintained for a period of 7 years. [24]

As thoroughly explained on StackOverflow [25], and referenced by a presentation by Google [26], Guava greatly benefits from the Java 5 features: generics, varargs, enums, and autoboxing. Generics is one of the reasons Google chose to create a new library instead of improving Apache Commons, as said here.

The code is full of best practices and useful patterns to make the API more readable, discoverable, performant, secure, and thread-safe. These patterns are everywhere in the code and they are not only *for the show*, they have a real value: the API is a pleasure to use, easier to learn, more efficient, many classes are simpler and thread-safe due to their immutability.

As for translating ACC to Guava, please refer to the Guava wiki here.

What's Next for Guava

Guava is still under active development, what are the plans for the future?

Plans for Guava 20 and 21

The big discussion for the last two years has been the discussion about how to support Java version 8. Complicating the process is that Java 8 includes classes that are very similar to Guava classes. Furthermore, Java 8 includes lambdas which are not supported yet by Guava and more importantly neither does Android. The crux of the problem is therefore that Guava needs to be updated while still holding on to the philosophy of compatibility of previous Java versions.

> We are confident that backwards compatibility with previous Guava versions is the best way forward for us and for our users. I'm hesitant to estimate a timeframe for an official release, largely because we're somewhat constrained by the rest of Google. [27]

In December 2015, it was announced that support for Java 8 would introduced soon.

> Guava 20 (to be released in early 2016) will be the final Guava release to support Java 6, or even Java 7. Guava 21 (ideally mid-2016) will require Java 8. So, if you're unable to upgrade to 8, you may be using 20 for a while. We don't expect to issue patch releases except for relatively serious issues. [28]

How does this affect Android?

> With version 21 we will also kick off a new fork, presumably called guava-android. It will maintain Java 6 compatibility, allowing it to be used for Android apps supporting a minimum version of Gingerbread. [28]

Usage Statistics and Phenomenon

In terms of usage in the development world, few Java oriented projects rival Guava. Just to get an idea as to how popular Guava is, an analysis that was done in 2013 [29] showed that it was the 8th most used external library from over 10,000 GitHub projects. In a further work done in 2015 [30], based on 40,000 Java projects hosted on GitHub, Guava is the 5th most used API.

Guava currently (March 2016) has 8150 stars on GiHub which according to Borgus et al [31] makes it minimally a top-10% project in terms of followers.

According to a study done in 2015 [32], Guava's most popular version (29% of the users) is 14.0.1 even though Guava 18 was the most current version at the time. Furthermore, no major version after 14 is in the top 5 most used versions. What is even more surprising is that 9.6% of the methods (and classes) are ever used by clients. Since Guava has an active depreciation policy, it seems likely that the remaining methods and classes are used inside Google for various projects.

In general, Guava's overall popularity is increasing but the current user base is slow to adopt new versions.

Conclusion

The Guava project is a library used to expand the Java Standard Library. In this chapter we studied the architectural structure of the project from multiple viewpoints and perspectives. Each of the following parts: context viewpoint, information viewpoint, the evolutionary perspective and the security perspective are highlighted and reported in the appropriate paragraphs of the current work. Moreover we actively contributed to the Guava project proposing a feature improvement in the source code aim to add a missing functionality in the Guava library.

The purpose of our project was to analyze the architecture of the Guava library from different viewpoints and perspectives. The first step we took was the stakeholders analysis. In this part we discovered that the main contributors are members of the Google team. The Guava project, like other libraries, does not have well defined stakeholders. They will be defined or inherited after the integration of the library in a project.

Then was the Context Viewpoint analysis in which we found the differences with JDK approach and the use of the Beta features. The third step was Development Viewpoint analysis. In that case we studied the module organization and the way how testing and design are done.

Next was the Information Viewpoint analysis in which we found a simplistic architecture. The reason is in the design of the library. The basic idea in the Input and Output case is manage the information in Source and Sink stream.

In the evolution perspective we found that the major changes come after the community discussion but at same time the team has an internal requests list to guarantee an high grade of quality. This approach is a starting point to guarantee a good security level. In addition due to the nature of the project the security is often an external problem linked to the host project. The Guava community simply does not have security as a priority for this reason.

The last step in our analysis we comapred Guava with the Apache Commons Collection project. Guava has some key advantages including a more frequent release cycle.

Finally, what did we learn from this experience? Guava is a well-established project with a concise philosophy and thriving community. Though the project is maintained by Google, the community is actively involved in its development especially in the form of discussions. These various stakeholders define many aspects of the architecture from how Guava processes data to its lack of security oriented features; they even define the development speed and style. Viewing a real-world project has given us a deeper appreciation for the architecture of software. Source code is just a fraction of the story, it is the entire ecosystem surrounding Guava that makes it the project it is.

References

1. "Google Core Libraries for Java 6+", GitHub, 2016. [Online]. Available: https://github.com/google/guava. [Accessed: 30-Mar-2016].
2. "Google Core Libraries for Java 6+ Wiki", GitHub, 2016. [Online]. Available: https://github.com/google/guava/wiki. [Accessed: 30-Mar-2016].
3. "com.google.common.base.Optional", Google.github.io, 2016. [Online]. Available: http://google.github.io/guava/releases/snapshot/api/docs/com/google/common/base/Optional.html. [Accessed: 30-Mar-2016].
4. "com.google.common.base.Splitter", Google.github.io, 2016. [Online]. Available: http://google.github.io/guava/releases/snapshot/api/docs/com/google/common/base/Spitter.html. [Accessed: 30-Mar-2016].
5. "com.google.common.collect.SetMultimap", Google.github.io, 2016. [Online]. Available: http://google.github.io/guava/releases/snapshot/api/docs/com/google/common/collect/SetMultimap.html. [Accessed: 30-Mar-2016].
6. "com.google.common.collect.Iterables", Google.github.io, 2016. [Online].

Available: http://google.github.io/guava/releases/snapshot/api/docs/com/google/common/collect/Iterables.html. [Accessed: 30-Mar-2016].

7. "Maven Repository: com.google.guava", Mvnrepository.com, 2016. [Online]. Available: http://mvnrepository.com/artifact/com.google.guava/guava. [Accessed: 27-Mar-2016].

8. "Search result of 'Guava' in GitHub", GitHub, 2016. [Online]. Available: https://github.com/search?q=guava. [Accessed: 27-Mar-2016].

9. "(AMA) We're the Google team behind Guava, Dagger, Guice, Caliper, AutoValue, Refaster and more -- ask us anything!", reddit, 2014. [Online]. Available: https://www.reddit.com/r/java/comments/1y9e6t/ama_were_the_google_team_behind_guava_dagger. [Accessed: 27-Mar-2016].

10. K. Bourrillion, "Guava for Netflix slides", googlecode.com, 2010. [Online]. Available: https://guava-libraries.googlecode.com/files/Guava*for_Netflix*.pdf. [Accessed: 27-Mar-2016].

11. "Artifacts using Guava: Google Core Libraries For Java", Mvnrepository.com, 2016. [Online]. Available: http://mvnrepository.com/artifact/com.google.guava/guava/usages. [Accessed: 27-Mar-2016].

12. "Guava: Google Core Libraries for Java 19.0 API", Google.github.io, 2016. [Online]. Available: http://google.github.io/guava/releases/19.0/api/docs/. [Accessed: 27-Mar-2016].

13. "Learn more about the Public Suffix List", Publicsuffix.org. [Online]. Available: https://publicsuffix.org/learn/. [Accessed: 27-Mar-2016].

14. G. Wielenga, "What is the Google Collections Library?", Javalobby.org, 2007. [Online]. Available: http://www.javalobby.org/articles/google-collections/. [Accessed: 27-Mar-2016].

15. "Google Java Style", Google.github.io, 2014. [Online]. Available: https://google.github.io/styleguide/javaguide.html. [Accessed: 27- Mar- 2016].

16. "IO Explained", GitHub, 2016. [Online].

Available: https://github.com/google/guava/wiki/IOExplained.
[Accessed: 30-Mar-2016].

17. "Collection Utilities Explained", GitHub, 2016. [Online].
Available: https://github.com/google/guava/wiki/CollectionUtilitiesExplained.
[Accessed: 30-Mar-2016].

18. "Caches Explained", GitHub, 2016. [Online].
Available: https://github.com/google/guava/wiki/CachesExplained.
[Accessed: 30-Mar-2016].

19. "Guava commit convention", groups.google.com, 2016. [Online].
Available: https://groups.google.com/forum/#!searchin/guava-discuss/anand/guava-discuss/gsgfYUJrxYk/jQzGbIxprVEJ.
[Accessed: 30-Mar-2016].

20. "Guavate - Guava and Java 8", groups.google.com, 2016. [Online].
Available: https://groups.google.com/forum/?utm_medium=email&utm_source=footer#!msg/guava-discuss/oWv4ee0BCHc/2UO4Yg2dqHgJ.
[Accessed: 30-Mar-2016].

21. "Pull Requests", GitHub, 2016. [Online].
Available: https://github.com/google/guava/pulls.
[Accessed: 30-Mar-2016].

22. "Hashing Explained", GitHub, 2016. [Online].
Available: https://github.com/google/guava/wiki/HashingExplained.
[Accessed: 30-Mar-2016].

23. "Releases", GitHub, 2016. [Online].
Available: https://github.com/google/guava/releases.
[Accessed: 30-Mar-2016].

24. "Apache Commons Collections", Mvnrepository.com, 2016. [Online].
Available: http://mvnrepository.com/artifact/commons-collections/commons-collections.
[Accessed: 30-Mar-2016].

25. "What are the big improvements between guava and apache equivalent libraries?", StackOverflow, 2016. [Online].
Available: http://stackoverflow.com/questions/4542550/what-are-the-big-improvements-between-guava-and-apache-equivalent-libraries.
[Accessed: 30-Mar-2016].

26. "An Overview of Guava Devoxx FR April 2012", guava-libraries.googlecode.com, 2016.

[Online].
Available: https://guava-libraries.googlecode.com/files/AnOverviewofGuavaDevoxxFRApril2012.pdf.
[Accessed: 30-Mar-2016].

27. "Guava and JDK 8 / Java 8", groups.google.com, 2016. [Online].
Available: https://groups.google.com/forum/#!searchin/guava-discuss/Java/guava-discuss/fEdrMyNa8tA/jxvkaPauPfkJ.
[Accessed: 30-Mar-2016].

28. "News about Guava and Java 8", groups.google.com, 2016. [Online].
Available: https://groups.google.com/forum/#!topic/guava-discuss/ZRmDJnAq9T0.
[Accessed: 30-Mar-2016].

29. T. Weiss, "We Analyzed 30,000 GitHub Projects - Here Are The Top 100 Libraries in Java, JS and Ruby", blog.takipi.com, 2013. [Online].
Available: http://blog.takipi.com/we-analyzed-30000-github-projects-here-are-the-top-100-libraries-in-java-js-and-ruby/.
[Accessed: 30-Mar-2016].

30. "Guice Getting Started", GitHub Wiki, 2014. [Online]. Available: https://github.com/google/guice/wiki/GettingStarted. [Accessed: 30-Mar-2016].

31. Sawant, A. A., & Bacchelli, A. (2015, May). A dataset for API usage. In Proceedings of the 12th Working Conference on Mining Software Repositories (pp. 506-509). IEEE Press.

32. Borges, H., Valente, M. T., Hora, A., & Coelho, J. (2015). On the Popularity of GitHub Applications: A Preliminary Note. arXiv preprint arXiv:1507.00604.

33. Sawant, A. A. (2015). fine-GRAPE: fine-Grained APi usage Extractor An Approach and Dataset to Investigate API Usage (Doctoral dissertation, TU Delft, Delft University of Technology).

by Andrea Lorenzo Pallini, Philipp Kogelnik, David van Es, Jaroslav Ševčík

Delft University of Technology

Abstract

Habitica is an open-source, cross-platform, role-playing game. It is designed to gamify reality to help build positive habits and organize daily personal activities. The goal of this project is to analyze the architecture of the software from different viewpoints. At first, we give an analysis of the stakeholders involved and the context view of the system. Next we analyze the development and information view and identify technical debt. Then we perform a feature analysis and from this describe the variability. Lastly we look at the level of Internationalization present in Habitica. In conclusion, we share our experiences with contributing to the project and look to the future of Habitica.

Table of Contents

1. Introduction

In February 2012 Tyler Renell wanted to limit his personal consumption of cigarettes, beer, coffee, junk food and excessive internet use. As a gamer, he felt that he could combine his struggle to curb his bad habits and his love of video games. He started working on a program that would allow him to track his daily tasks, build good habits, and reward him for good behavior by treating life as a video game: thus Habitica was born. An early open-source version went live on GitHub at the beginning of 2013. Over time, as interest accumulated, the project grew and Tyler was joined by several other enthusiasts to form the core of Habitica's staff. To boost the project further, Tyler started a crowdfunding campaign and raised almost $50K - twice the target amount. Nowadays Tyler has accepted the CTO role, leaving the coding to others. Habitica is still going strong and growing, with an active developer community and hundreds of thousands of players.

Habitica is inspired by the popular fantasy role-playing game genre. When they first start the game, a new player creates an avatar, a virtual representation of themselves. By fulfilling user defined tasks, they are rewarded with gold and experience. This allows them to improve their avatar with new specializations, abilities, equipment or buy a cute pet. They can also form a party with other players to co-operate with them to fight terrifying monsters, like the infamous Basi-List, together. There are also many guilds which players can join to debate with like-minded Habiticans or get advice from more experienced players or even the staff members themselves. For a more exhausting list of game features visit their web page [10].

2. Stakeholder Analysis

In this section we take a look at the people, systems and organizations involved in Habitica and categorize them according to the Rozanski & Woods book Software Systems Architecture [1], with some additions. This book serves as the basis for most of the analysis we have done. The stakeholder's names we use throughout the document refer to their in-game or GitHub nicknames.

The staff

Lefnire	Lemoness	Redphoenix	SabreCat	Paglias	Viirus
Original creator, CTO	Community manager, runs social media	CEO, legal	developer	developer	mobile developer

The staff is the main organ of the Habitica project. They are responsible for decisions about the present and future, solving legal issues and raising the money for the survival of the game. The staff consists of the six people in the table above: Lefnire, Lemoness, Redphoenix, SabreCat, Paglias and Viirus. The particular relationships of the members to the types of stakeholders can be found in the next section.

Stakeholders

Role	Stakeholders
Acquirers	Lefnire (Tyler Renelle), Redpheonix (Vicky Hsu)
Assessors	Redpheonix
Communicators	Lemoness (Siena Leslie), Breadstrings, Alys
Support	Lemoness, Redpheonix, Alys, SabreCat (Sabe Jones), Viirus aka vliRuS (Phillip Thelen), Blade aka Crookedneighbour (Blade Barringer)
Moderators	Lemoness, Redpheonix, Alys, beffymaroo, Breadstrings, Cantras, deilann, Megan
Suppliers	MongoDB, ExpressJS, AngularJS, Heroku, Amazon Web Services
Developers, Maintainers	Lefnire, SabreCat, Paglias (Matteo Pagliazzi), Viirus, Blade, Alys, Lemoness
Users	General public, Companies, Premium users
Funders	Kickstarter backers, Donators, Premium users

Acquirers oversee the procurement of the system or product. As the original creator of Habitica and founder of the Kickstarter campaign **Lefnire** can be classified as an acquirer. He was responsible for raising the funds required to make the entire project possible. When Lefnire became CTO of the project **Redpheonix** took up the mantle of main CEO.

Besides being the CEO Redpheonix is also the main **assessor** of Habitica. Assessors oversee the system's conformance to standards and legal regulation. Redpheonix is the legal advisor who 'handles all of the terrifying paperwork and business planning'.

A very important group of stakeholders for satisfying existing user and the acquisition of new users are the **communicators**. They explain the system to other stakeholders via its documentation and training materials. There are several communication channels like wiki, Facebook, Twitter or email. The wiki page is the main channel used by Habitica for learning about the gameplay and introduce the project to new users and developers. Habitica is also a game with an emphasis on friendly social interaction. For this purpose, the in-game Tavern and the Newbies Guild in particular are also top learning resources, where new players can ask all the questions they want without fear of being ridiculed. The administrators of the wiki are the main communicators, as is the Habitica staff and all players who actively participate in the guilds.

The group of communicators is partially coupled with the **support** staff of Habitica. This group provides support to users for the product or system when it is running. Some of them are helping mainly with in-game content - Lemoness, Redphoenix, whereas others are focusing on technical issues - Alys, SabreCat, Blade, Viirus. It can be seen that Lemoness and Alys are reponsible for giving support to existing users and communicating the game to the outside. They use many different channels for communication like GitHub, in-game chat or email.

Another type of stakeholders related to the communication in the game are the **moderators**. This group supervises the communication in Habitica. They have the right to delete inappropriate posts. They are also recognized persons and their advice can have an influence on many staff decisions.

Suppliers for this project build and/or supply the hardware, software, or infrastructure on which the system will run. The software and technologies used to build Habitica are roughly divided into five sections: Server-side, Client-side, Testing, Services, and Other (mobile developing and GitHub). External services like the hosting provider apply some constraints regarding availability to the project. Habitica has no internal hardware resources. For this reason, they highly rely on those external services. If there are downtimes at these services the Habitica system is not able to run any more.

Then there are the stakeholders who are responsible for implementing the features which should excite the users. The role of the **developer** is to construct and deploy the system from specifications and **maintainers** manage the evolution of the system once it is operational. Habitica is an open source project and the lines between the developers and maintainers are blurred. There is a core staff who is ultimately responsible for all these issues with their roles overlapping. Habitica is based on a concept which potentially allows everyone to contribute to it and become a developer. However, some of the staff members and few others are the ones who do most of the work from designing, coding, testing to acceptance (merging of pull requests)[4][6]. The team also manages the main evolution of the system, for this reason they also belong to the maintainers group. The staff members are not the only "good stakeholders"[1] in the system, there are also a few very active contributors who support the staff in development and maintenance. They acquired information and experience during the collaboration period ("informed"), and have also acquired the responsibility to make important and difficult decisions with the right to speak on behalf of Habitica or merge pull requests ("committed" and "authorized").[7][8] The developers responsible for accepting new contributions are the first five people in the table. Lemoness is the main pixel artist responsible for the appearance of the game.

Very important stakeholders for an exciting game are the **users** themselves. They are the people making use of the system, so the functionalities are designed to satisfy their needs. They also have the opportunity to request new features. The users come from the following groups: The public: people who may have problems with daily tasks, time management, bad habits and like fantasy RPG. Companies: They can encourage(force) employees to use Habitica in order to increase their motivation and productivity. Families, health and wellness groups: leaders of these groups want to help their members change their behaviors. Premium users: users that want to support Habitica in exchange for in game currency.

The group of users highly overlaps with the **funders** of the project. Funders are users who support Habitica with spending money on the project. This includes all the people who backed Habitica through the Kickstarter campaign, the premium users and the donators.

Power/Interest Graph

Each and every stakeholder has a different level of interest in the project, and a certain amount of power to influence it. The stakeholders can be categorized into the following four groups:

- **Low power and low interest**
    - The power of common users is limited but still they can convince the staff (through in-game chat or GitHub issue) to change or add functionalities.
- **Low power and high interest**
    - Premium users are very interested in Habitica, they are strictly involved in the game. Developers are active in discussions but they still need the staff approval.
- **High power and high interest**
    - The staff is part of this group.

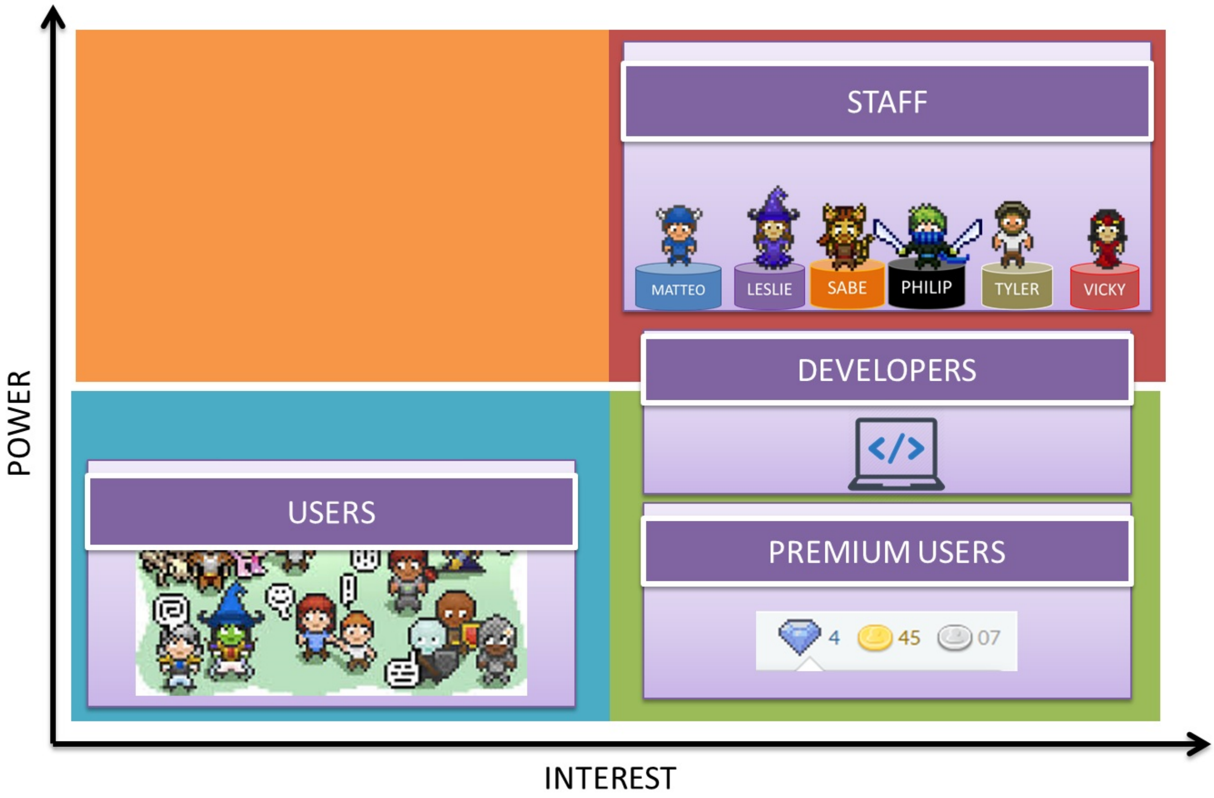


Figure 1: Power/Interest graph

3. Context View

Habitica is reliant on and communicates with many different systems. The context view model gives an overview of the most important systems that Habitica interacts with. The purpose of this view is to see the big picture: how does Habitica fit in to the larger universe around it.

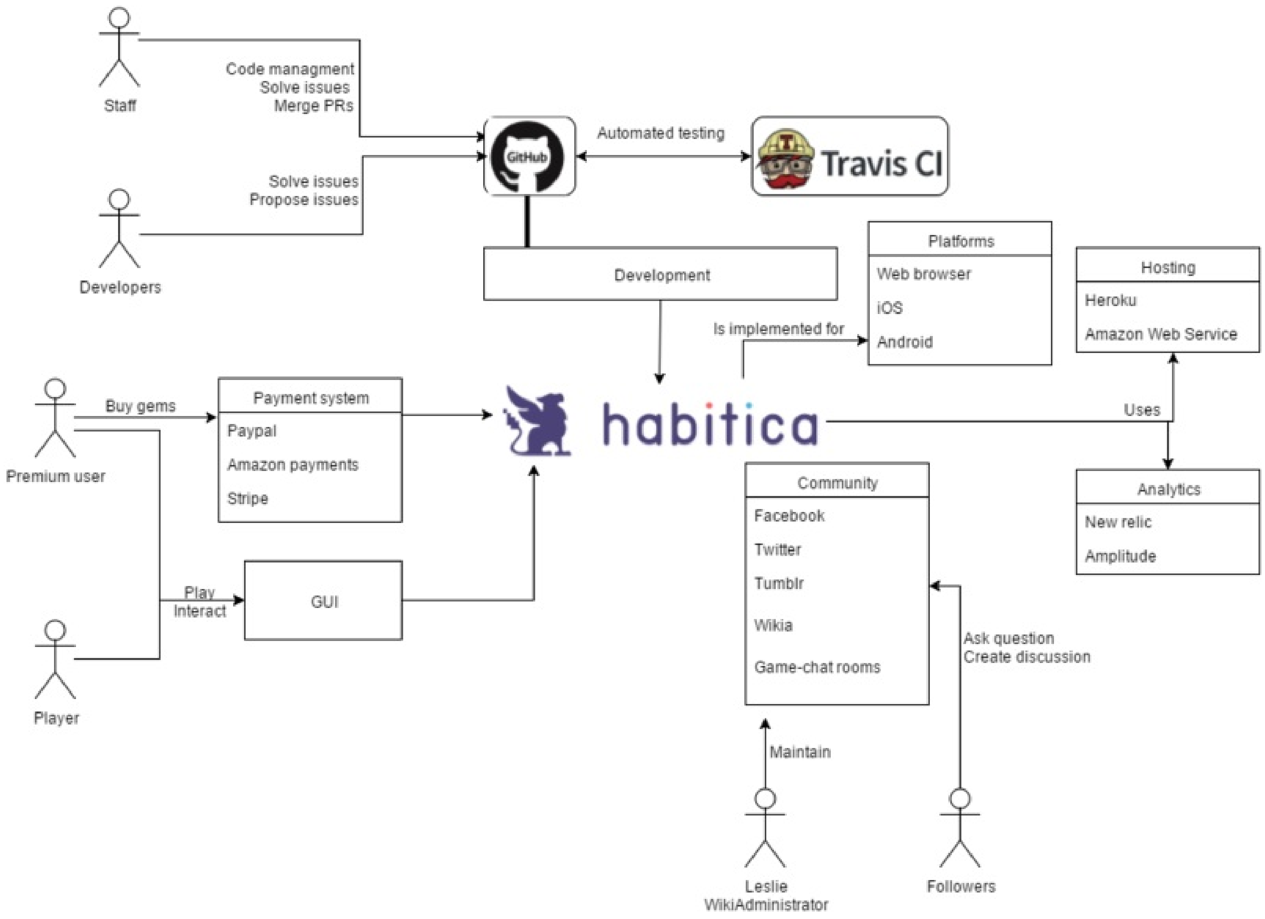


Figure 2: Context Diagram

A major platform for the development process is GitHub. It is used for code management, solving issues and merging pull requests for updates or bug fixes. GitHub is also used as a customer support platform: users with problems are encouraged to create an issue for any bugs or defects they may encounter while playing. TravisCI complements the functionalities of GitHub. It is used as a continuous integration platform and is the main platform for performing tests.

In addition to the GitHub issues a big part of the communication with the community goes through either in-game chat rooms, the wiki, or through their social media pages. Habitica has pages on Facebook, Twitter and Tumblr. In addition, Facebook is also used as a login provider, allowing users to use their facebook accounts.

Habitica uses Heroku and Amazon Web Service (AWS) as hosting service providers for the whole project to handle all 1,000,000 players. In the beginning they just used Heroku, but in the moment they are moving the project towards AWS. For tracking the performance of the system and the user interaction there are two service providers which are used in this project: New Relic and Amplitude.

Habitica also provides public API for the creation of external extensions. Therefore, there can be many external applications (like the Data Display Tool) interacting with the system. The exact nature of these interactions depends on the nature of the extension itself.

4. Development View

Module structure model

The Development view attempts to capture all aspects of the system relevant to the development of the project. Habitica is composed of two main components: the client-side and the server-side. These components communicate via an API provided by the server-side.

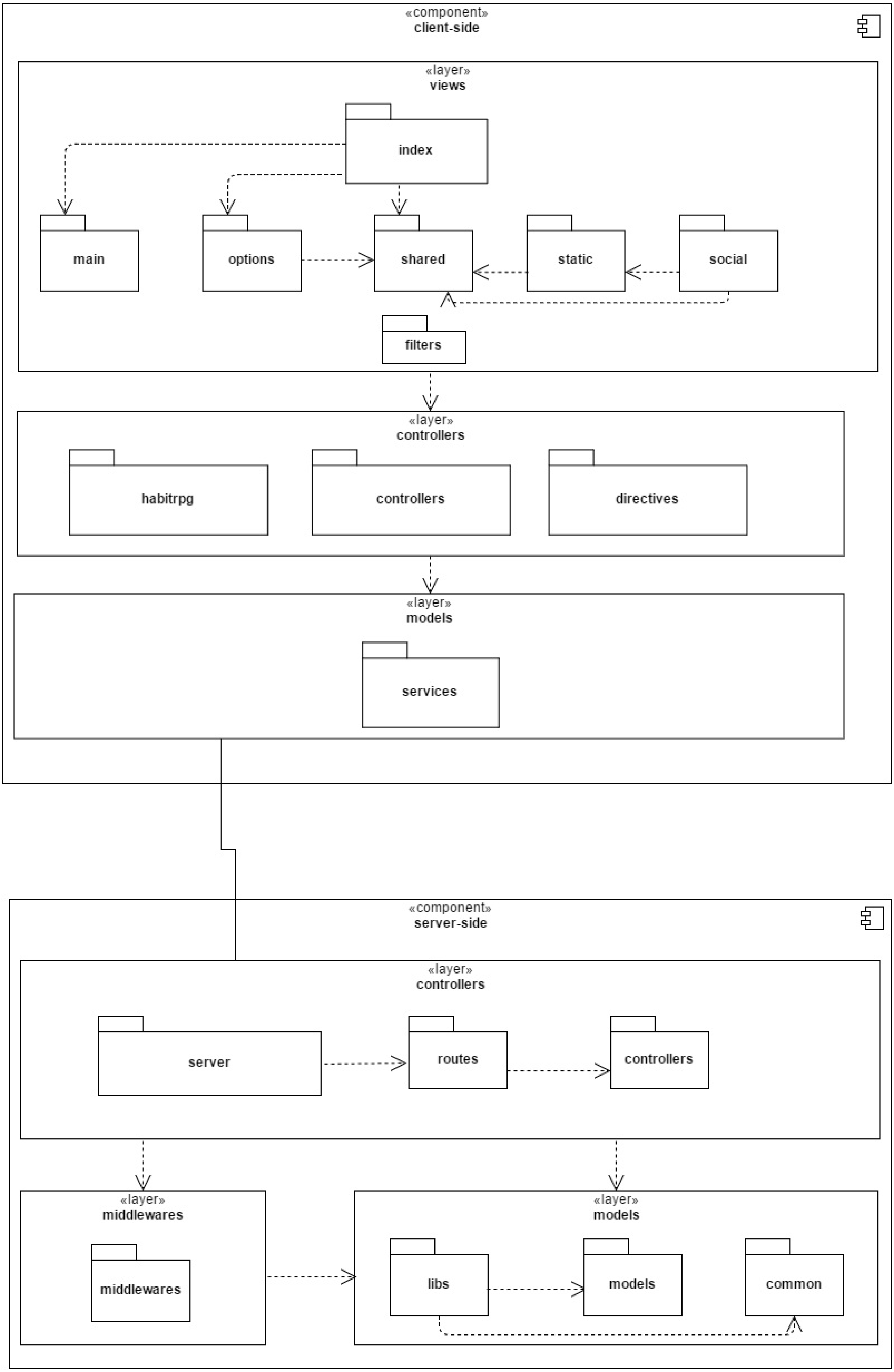

«component»
client-side
«layer»
views
index
main
options
shared
static
social
filters
«layer»
controllers
habitrpg
controllers
directives
«layer»
models
services
«component»
server-side
«layer»
controllers
server
routes
controllers
«layer»
middlewares
middlewares
«layer»
models
libs
models
common

Figure 3: Development view structure model

The code on the server-side is divided into several layers and modules in accordance with the framework used (Express JS). In the “controllers“ layer there are mainly modules dealing with routing. In the “middlewares” layer there are modules dealing with preprocessing of requests and responses. In the third layer, “models”, there are the business logic, utility libraries, handling of database access and some shared resources useful for the multiple Habitica repositories. All the dependencies go in the direction controllers -> middlewares -> models.

The code on the client-side is divided as a pure MVC in accordance with the Angular JS framework. This pattern separates the view from the business logic, which increases the reusability of code. The “view” layer contains the Jade files. These files are compiled into the HTML code. We also added the filters scripts here, because they only transform the displayed data a little (for example displaying the date properly). The “controllers” layer contains one explicitly declared Angular JS module called habitrpg. All the other “modules” are defined as an extensions of it. These dependencies are omitted from the diagram for clarity. These modules are responsible for routing and dynamic rendering of the parts of the view. The services represent the “model” layer here. They are substitutable objects used to share code across the application and they handle the communication with server-side.

Common design model

There are some parts of the system where common processing rules are imposed. The first and probably one of the most important points from the perspective of the system users is the internationalization. All strings in the system must be localized and stored in the locale files in the directory "common/locales/". Another principle of the project that can be seen after a closer look at the source files, is to use an existing framework whenever possible. If this principle is satisfied the developers can fully concentrate on the core features which cannot be covered with external libraries. For the database there should be model classes which are able to hold the data, store it and retrieve it.

The concrete implementation of external libraries should be hidden to the developers. In the project this is done with the facade pattern [12]. This ensures that the developer can use the functionalities of external libraries without having to care for changes of the interfaces if one of the libraries gets exchanged by another one. The logging part of the software is done with different logging providers. For this reason the common interface is not implemented as a facade, but as a mediator [12]. The mediator aggregates the functionality of different logging providers and makes it easier to add new ones without influencing other parts of the code.

The most important design approach for the whole project is the usage of the model-view-controller (MVC) pattern.

There are also some parts of common software used in this project. For the separation of the layers on the client side the framework AngularJS is used. The database interaction with MongoDB is done with an object data mapper (ODM) called Mongoose. For the internal logging the Winston framework is used, which can be extended to log to many different destinations (database, plain text file, xml file, ...). For the internationalization part the self-written i18n functionality should be used. It parses the translations directory and stores the translations for the user language in the memory. This component ensures a fast retrieval time for the needed texts.

Configuration strategy

There are several different approaches to configure the application. The more technical configurations like logging or choosing an analytics provider, that serves particularly to developers and operators, are managed via a configuration file or by commenting parts of code. The setting of user preferences is handled through a database variable that points to a relevant file on a disk. Then there are various intrinsic or third party extensions that are installed separately from the game, as a browser add-on, web page widget, bookmarklet or desktop application, and cooperate with it via provided API. Other types of extensions just amend the appearance of the web site without any direct interaction with it. They usually need the Stylish[11] web browser add-on. To differentiate the appearance of the web application on desktop and mobile devices, CSS media queries and JavaScript conditions are used. Lastly, for some mobile devices (Android, iOS) there are native applications replacing the client side of the system completely. However, they lack many features of the web application (as of March 3, 2016).

As identified above, a variety of different mechanisms is used to add variability to Habitica. After a closer analysis of pull requests and issues on GitHub, we found that the variability mechanism didn't change over time, but it differs among different developers. Some configuration possibilities were not part of the initial versions, but were added in later stages of the development.

5. Information View

The purpose of any information system is to manipulate data in some form. Habitica stores data persistently in a database managment system using MongoDB. MongoDB is one of the most popular NoSQL databases.

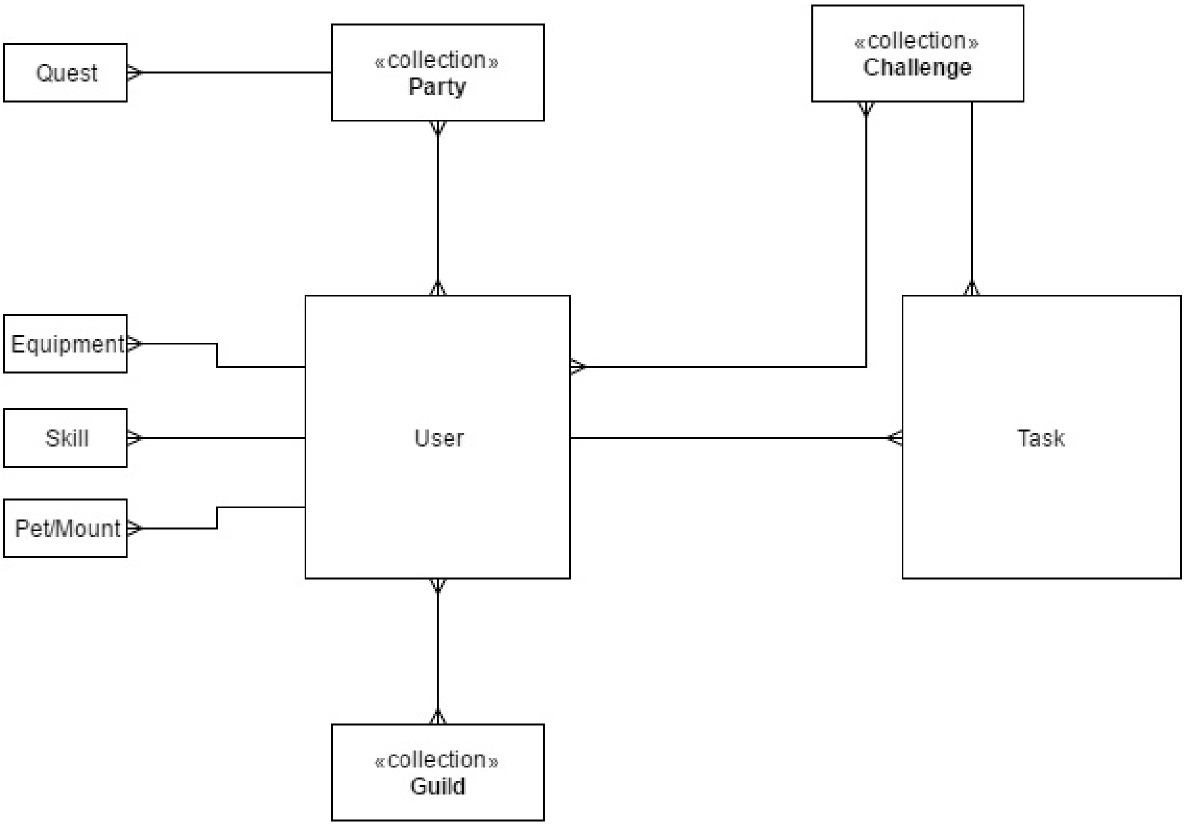


Figure 4: Static information structure model

The diagram presented in Figure 4 shows the logical relationship between the principal entities in the game. The two main entities are the user and the task.

The user is personalized by the equipment chosen, the skills developed during the game and his own pets/mounts. Moreover, the user can create or participate in parties and guilds with their friends.

The tasks are the most important element in Habitica. Every user must complete them in order to increment his own game-level and earn gold. Each task is associated with a single user. Habitica gives the possibility to create challenges among the players. A challenge is a collection of tasks which are automatically assigned to all the participants.

However, the logical view mentioned above doesn't directly map to the database structure. It is partially because of the use of the NoSQL DB, which doesn't have a fixed structure. Another reason is the use of a second storage - a JavaScript file. We will explain the persistent representation of the main entities and the relationships among them.

In the database there are three core document types: User, Group and Challenge. The User document contains all of the information related to a concrete player. It contains the user information, tasks, customization references, gold, equipment, pet references, UI preferences, private messages, etc. The Group document represents all the parties and guilds with their chat logs and quests. The Challenge represents a group of tasks for the participants to fulfill, the list of participants themselves and the prize for the winners.

In addition to the database storage there is information stored in JavaScript files. In these files all of the various items and appearances you can have as a player are stored. For example, the equipment, spells, haircuts, etc. These items are then referenced from the database document.

Notable Characteristics

How Habitica deals with old data

With the increasing number of users the staff of Habitica worked a lot on the improvement of server's performance. One key issue is the reduction of database storage requirement. This is the main reason why Habitica does not keep all historical data for all tasks. Instead, older data are averaged and only the average is included in the data exports (data export is a feature which allows players to retrieve data about completed Habits and Dailies, as well as personal user data). The further back in time the data goes, the more data is combined together into a single average. For example, a data export might include several data points for the current week, one data point for each of the previous few weeks, one data point for each of the previous few months, and only one data point per year for previous years. As a (probable) side effect of this, when players examine their exported data for Habits, they will see at most one data point for each Habit per day, i.e. they will not see every click they made on the Habits' plus and minus buttons. This has been logged as issue 3079 but it is not currently known whether this behavior is deliberate or a bug. In addition, for non-subscribers, completed To-Dos are automatically deleted after 30 days to regain database storage space. For subscribers, completed To-Dos are automatically deleted after 90 days. If a player uses the "Delete Completed" button in the To-Dos' Completed tab, all completed To-Dos will be permanently removed. Because older task history is severely limited, players who want to keep a full history of tasks should export their data regularly.

Static Information Storage

One of the most interesting parts in the information flow is the transfer of static information to the database. All information about equipment, quests, pets and spells is stored as JSON objects in JavaScript files which are located in */common/script/content/*. When a user buys equipment, the information from the file is set as a reference in the user's document in the database. For example if a user buys a new instance of a shield, the information of the shield is looked in the related JavaScript file. Then the identifier of the shield is copied to the database entry of the user. The identifier includes the class of the user (rogue, warrior etc.), the type of the equipment and a sequential number. The attributes of the equipment, like strength boost, are still looked up in the JavaScript file.

A similar process where the data is copied to the database entry of the user is the challenge system. A user can create a challenge which consists of different habits, dailies, todos and rewards. Other users can join these challenges and get the included tasks transferred to their personal task list. The challenges are stored in a separate document collection in MongoDB and include all the information which is needed. When a new user joins the challenge, the information from the challenge is copied to the database entry of the user. This means that the personal task is independent from the challenge task. There is just a reference to the challenge kept to keep track of the challenge progress.

6. Internationalization Perspective

In order for a system to become truly international it must not overly depend on any one language, country, or cultural group. Habitica is a game that could potentially be enjoyed by people from all cultures or different parts of the world. To make sure that the absolute widest range of people is targeted there are some common concerns that need to be looked at.

A place where a person interacts with the system is called a touch point. To obtain a good level of internationalization it is first necessary to go through the system and locate all the touch points. Then, for each touch point each concern can be checked. This gives a good indication of what the current level is and also how much work is yet to be done.

The concerns we will address are the support for the following items:

- Multiple character sets
- Differently oriented text presentation
- Multiple languages
- Different cultural norms (i.e. units of measurement)
- Financial differences
- Cultural neutrality

Figure 5: Touch points

Touch Point Analysis

Habitica's main touch points are the game itself, the wiki, GitHub and Trello. For the game this would include all screens, buttons, pictures, dialog boxes etc.

The Wiki is available in six different languages: English, German, Spanish, French, Polish, Brazilian Portugese, and Russian. However, the English version contains much more information than the wiki versions offered in other languages. A lot of new users will come into contact with the Wiki and will use it to learn game basics. Therefore the most important pages, such as how to play guides etc., are given priority.

Language	Pages (as of March 30, 2016)
English	576
German	91
Spanish	206
French	141
Polish	3
Brazilian Portuguese	34
Russian	143

The game itself offers support for a lot more languages. There are translations in 24 languages (including the much overlooked Pirate English), in various states of completion. In some cases the translators have not yet completely finished their work and the American English version overrules the translation. Also, as Habitica has frequent updates and on top of this also has seasonal in-game events, it is unlikely that the newest content will have been translated when it is just released.

The game and wiki all support multiple character sets and has no problem representing symbols such as icons or chinese texts. Habitica tries to take into account different cultural norms. For example, the way dates are represented vary across the world and this can be customized to match the players preference.

The financial differences are not completely taken into account. The prices for gems are still all listed in American Dollars. However, these differences are mostly resolved through the use of the external payment providers PayPal, Amazon or Stripe which will automatically give the conversion.

For the developer side of things there are two main touch points: GitHub and Trello. Historically, English has been the go-to language in computer science related fields with the most popular programming languages using English naming conventions for keywords and the like. Likewise, in Habitica all the code is in English. Furthermore, all discussions, issues, bug reports and feature reviews are typically done in English.

Like with many open source projects getting things done essentially relies on enthousiastic hobbyists to put in the required effort. Such is also the case with all of Habitica's translations. That means that there is somewhat of a chicken and egg problem in that users of Habitica who would like to see it in their native language must first help to translate it.

7. Contributions

Habitica is a project which relies on help from external contributors. The manpower in the staff is limited, and they rely mainly on volunteers. For this reason, the staff is very welcoming when new people want to contribute. Work can be done in numerous different ways. Writers can update the wiki entries or translate the game in new languages. Designers can create new images for new equipment or pets. Software developers are able to create new features, fix bugs or contribute to the test coverage. But also people without knowledge in those areas can contribute to the project in suggesting new features or reporting bugs.

The main focus of our team is the software development part of Habitica. In the beginning it can be very hard to find a starting point for contributions in such a big project. Fortunately, the staff and main contributors of Habitica label all features according to their importance and complexity. With these labels we found some issues on which we could work on. We provided a pull request which got updated some days later (#6736). We were added to the Hall of Heroes of Habitica and got the Tier 1 contributor award. This is a very interesting approach for motivating contributors and reward them for their work.

Our confidence grew after this first pull request and luckily we found our own bug in the project just a few days later. We opened the issues and started working on it by ourselves. A few days later our second pull request got merged (#6740).

One big problem of the Habitica project is the low test coverage for client side code. According to coveralls.io, just 52% (as of March 30, 2016) of the code is covered by different tests. This could be a serious problem when new features are introduced, because nobody knows if the old features are still functional. For this reason, we wanted to create new unit tests for the project, to increase this metric. After a closer investigation of the project, we found some rudimentary experiments with end to end tests with the Protractor framework. Our focus switched a bit and we wanted to focus on creating a test suite which is based on the page object pattern [9]. As of March 26, 2016, we are still working on the test suite to make it compliant to the testing strategy of the Habitica team. (#6868 and #6876)

8. Conclusion

In this project we have analyzed Habitica from a Software Architecture perspective. We have found that Habitica is built and maintained by a core set of staff who fulfill multiple roles and are largely responsible for overseeing the whole project. Like many open source projects, they rely heavily on external contributors: enthusiastic volunteers who are willing to help them fix bugs, add features, and help out in general.

The architecture is stuctured nicely into different layers with a single page application for the client side and backend services accessible through an API. They develop the project using very flexible technologies that allow the developers to quickly make changes to existing features. For example the use of MongoDB and JavaScript. They also put great effort into making the process of getting involved as easy as possible for new developers by using popular technologies and providing "Getting Started" guides. This was especially helpful to us as it allowed us to jump in without too much effort.

The biggest issue that we encountered was the low test coverage. Our group tried to improve this by adding our own end to end tests for the tasks page and the user customization features.

The future looks bright for Habitica. They are currently at the next stage of developement with the imminent introduction of the API v3. With this, and the increased focus on improving testing and code quality the overall quality of the application will likely become even better.

We are glad to have been able to get some insight into a big open source project and are very happy to have contributed to Habitica. We hope that Habitica continues growing and that thousands of players keep getting the most out of the gamification of life!

9. References

1. Nick Rozanski and Eoin Woods. (2011). Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives. Addison-Wesley Professional.
2. Habitica staff. Guidance for Blacksmiths. http://habitica.wikia.com/wiki/Guidance_for_Blacksmiths, 2016
3. Wikipedia. Habitica. https://en.wikipedia.org/wiki/Habitica, 2016
4. Habitica staff. Staff presentation. http://blog.habitrpg.com/who, 2016
5. Habitica staff. User plans. https://habitica.com/static/plans, 2016
6. Habitica staff. Staff. http://habitica.wikia.com/wiki/Staff, 2016
7. Habitica staff. Community guidelines. https://habitica.com/static/community-guidelines, 2016
8. Habitica staff. Moderators. http://habitica.wikia.com/wiki/Moderators, 2016
9. Arie van Deursen. Beyond page objects. http://avandeursen.com/2015/06/25/beyond-page-objects/, 2016
10. Habitica staff. Habitica. https://habitica.com, 2016
11. Jason Barnabe. userstyles.org. https://userstyles.org/, 2016
12. Gamma, E. , Helm, R., Johnson, R., Vlissides J.: Design Patterns: Elements of ReusableObject-Oriented Software. Addison-Wesley, 1996

Karma: the JavaScript test runner

Figure 1. *Karma Logo*

By: **Lu Dai**, **Wesley van der Lee**, **Mourad el Maouchi** and **Liu Yiran**
Delft University of Technology, 2016

Abstract

Building online applications coincides with continuous testing. Testing these applications on all available browsers and platforms seems cumbersome. Karma stimulates Test Driven Development (TDD) of JavaScript applications by providing a productive testing environment where developers can just write the code and get instant feedback from their tests. In order to facilitate this environment a lot of logic requires to be implemented into the Karma system. This chapter studies the Karma system by discussing different viewpoints and perspectives on Karma's environment and internal structure. We discovered that Karma has been built by a large community of developers, which now also includes us.

Table of Contents

Introduction

Web applications have become a part of the average person's everyday life. They fulfill an unlimited number of purposes, which may range from social interacting through invoicing or playing games. Due to its flexibility and browser support, JavaScript has become one of the dominant languages to create these applications in. Building online applications coincides with continuous testing and hence requires a way to easily enable a Test Driven Development (TDD) approach, to safeguard the correctness of the implemented applications. This is where Karma comes into play. Karma supports this TDD approach by providing a productive testing environment where developers can write the code and immediately see the test results.

In this chapter, we provide a comprehensive architectural analysis of the Karma system. First, Karma is explained in short. Next, relevant stakeholders and their interests will be identified. Afterwards, the environment of Karma is discussed. Next, the functionalities provided by Karma are introduced with afterwards the features supporting these features. Moreover, a development view is provided with the technical debt for the system. At the end, a performance perspective is provided with a conclusion regarding the architectural analysis.

What is Karma?

Karma is *the Spectacular Test Runner for JavaScript*, which in essence is a tool that allows test execution, of JavaScript applications, in real browsers. Karma itself is also written in JavaScript. Karma started off as a Master Thesis project by Vojta Jína, carried out at the *Czech Technical University* in Prague.

The main goal of Karma is to enforce TDD by providing a productive testing environment to developers. In this environment, the developers should not have to set up tons of configurations, but rather a place where developers can write test code and get instant feedback from their tests. Getting rapid feedback, is what makes a developer productive and creative at the end.

From the official website of Karma[2], explaining the purpose of Karma:

"Things should be simple. We believe in testing and so we want to make it as simple as possible."

Stakeholder Analysis

Stakeholders are an important element for every software system. This also equals for Karma. Within the Karma environment, there is a various amount of stakeholders. Each of these stakeholders is concerned with one, or multiple, aspects of Karma. To be able to understand the architecture and the current state of the Karma system, firstly, the stakeholders and their interests and influences are analyzed.

In the first subsection, the identified of the stakeholders is elaborated on. Next, the interests and influences of those stakeholders are discussed. In addition, we discuss the stakeholders in the early stage and in the current stage of Karma.

Identification

Instead of identifying the stakeholders as individuals, they are grouped together and categorized. The categorization is based on the 9th chapter produced by Rozanski and Woods [1], which states that software systems have corresponding assessors, communicators, developers, maintainers, suppliers, etc. In the following table, Table 1, the stakeholder types are provided with the corresponding stakeholders in Karma.

Stakeholder Type	Identified Stakeholders
Assessors	This exists out of the community that discusses if an issue or pull request is in line with the standards. This can not be categorized to a few individuals, as the community is too large for this.
Communicators	Friedel Ziegelmayer, with the help of the community.
Developers	There is a total of 219 contributors (as of 29th of March) of the Karma project. The main ones are Vojta Jína, Friedel Ziegelmayer and Christian Budde Christensen.
Maintainers	Greenkeeper, Friedel Ziegelmayer and Christian Budde Christensen. Greenkeeper keeps updating plugin version, Friedel manages most issues and PRs and Christian was invited to become a maintainer due to his rich amount of contributions.
Suppliers	The compatible browsers, plugins and additional tools that Karma uses.
Support staff	Friedel Ziegelmayer. However, the community that uses and helps develop the project can also be seen as the support staff.
Testers	For every new feature, the developer who created that feature has to submit tests to it, before it gets merged. However, Friedel Ziegelmayer has the responsibility to create the main tests.
Users	There is a huge amount of users that make use of Karma. jQuery, AngularJS and a lot of internal Google Projects use it. This has been told to us by Friedel.
Community	Two communities are present for Karma: Karma Users and Karma Runner.

Table 1. *Stakeholders.*

Interests and Influences

As mentioned before, Karma has started off as a Master Thesis project. Hence, during the early development, the stakeholders of Karma differ compared to the current situation.

Early Stages

During the early stages of Karma, there were mainly *three* stakeholders. One of which were the JavaScript developers the system was made for. The other two stakeholders were Jína himself and his supervisor from the university. In that stage, it was easier and relatively faster to make changes that satisfied all the needs of the stakeholders. The JavaScript developers did not practice any influence to the development but were taken into consideration. The supervisor, and Jína himself, were the only stakeholders with influences to the development of the system. In the current stage, this is very different.

Current Situation

In the current stage, Karma is highly dependent on the open source community. The stakeholders that perform a type of influence and have their interest in the system are identified in Table 1. The user community has a very high interest in the functional correctness of the system, as they primarily use it. This user community exists from regular persons to entire organizations such as Google, with their AngularJS project, as some internal projects.

To depict the interest and influences of these stakeholders, they are best depicted in the form of a Power-Interest Grid (see Figure 2). This grid contains the important stakeholders and their interests in the system.

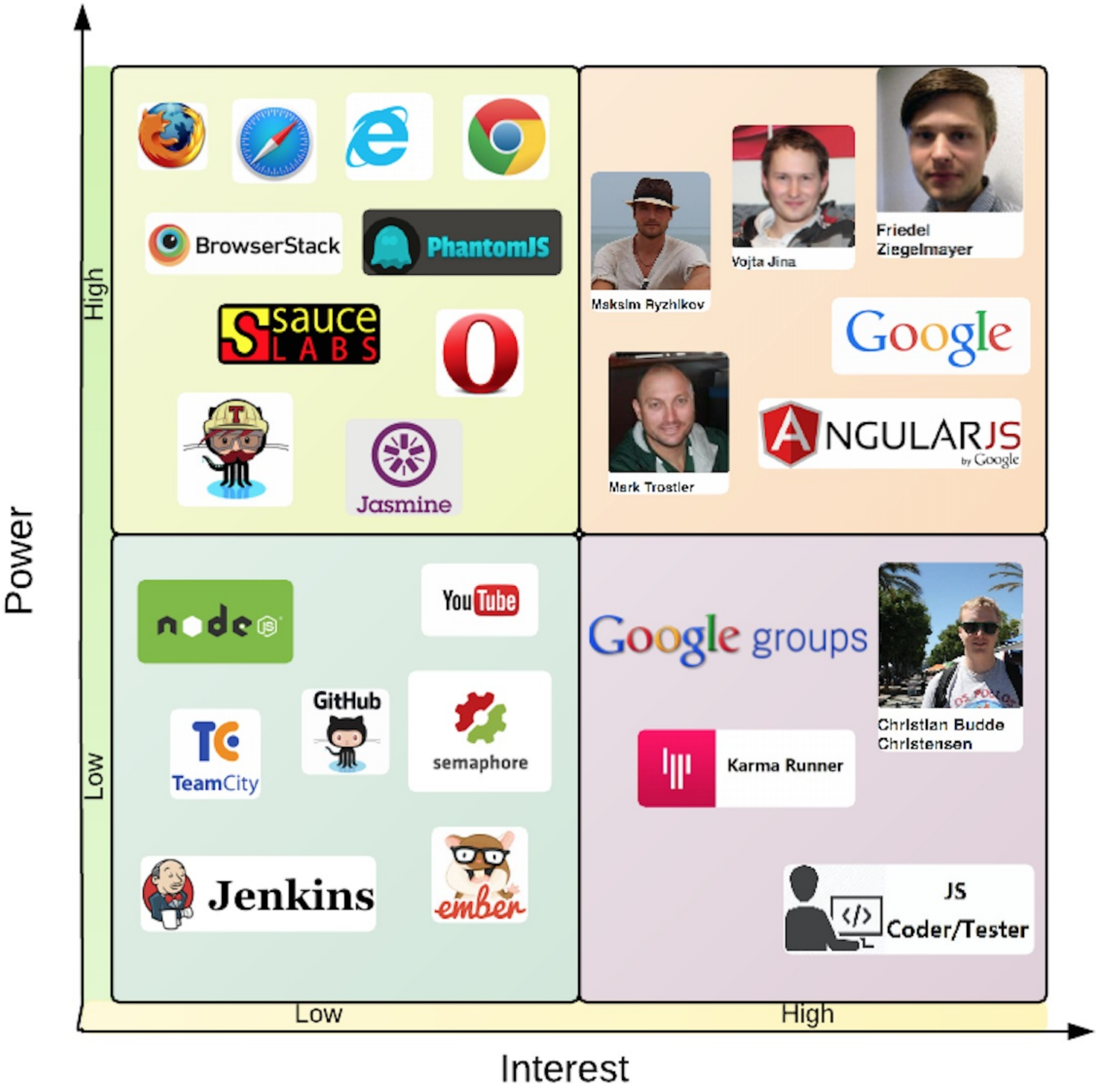


Figure 2. *Power-Interest Grid*

On the bottom-left corner of the Power-Interest Grid are tools that are used by Karma for the own development of it. It is hosted on GitHub and uses Continuous Integration tools. Next, in the top-left corner are the plugin suppliers which have low interest in Karma, but have high power. This results out of the fact that Karma launches real browsers and makes advantage of various testing frameworks to execute the tests. On the bottom-right corner, the general community and Karma users are depicted which has high interest, as they aid in the development and wanted tools, but do not have high power. The last group, on the top-right corner, contains the main developers and the two organizations that have both high interest and high power.

Environment Analysis

The stakeholders are identified with their corresponding interests and influences. In this section the environment within and surrounding Karma is discussed to gain more understanding. Next, the impact on the environment, during the development of Karma, is discussed.

In order to better understand the influences and impacts on Karma, or even from Karma on the surrounding environment, it is important to gain knowledge of what that environment consists of. The environment analysis has been separated into two parts, the internal environment and the external environment.

Internal Environment

The internal elements can be separated in three layers. These layers consist of the *client*, *library* and *server* layer. Below, the three layers are discussed.

- **Client:**

  Most user actions are available through the client layer where a user can carry out command line operations, web browser testing and debugging. The tests are performed in real browsers.

- **Library**:

  The library layer acts as a bridge, connecting the client and server. Karma uses a considerable amount of internal and external libraries to guarantee its performance. The reporter and logger provide relevant information and the gathered test results to the client. The middleware delivers the needs from the clients to the servers, such as informing the server to run, stop or perform a retry. Socket.io is one main external library used in Karma to guarantee web browser connection. The watcher keeps an eye on the files, specified in the configuration, and signals Karma to run with the help of the middleware once one of those files changes.

- **Server:**

  The Karma server controls, runs and stops processes. The runner can perform testing based on the configuration file or command line options. The latter are delivered by middleware. When the server runs, the library supports it with necessary files and the tests are run in the client layer.

Figure 3 depicts the internal environment of the Karma system.

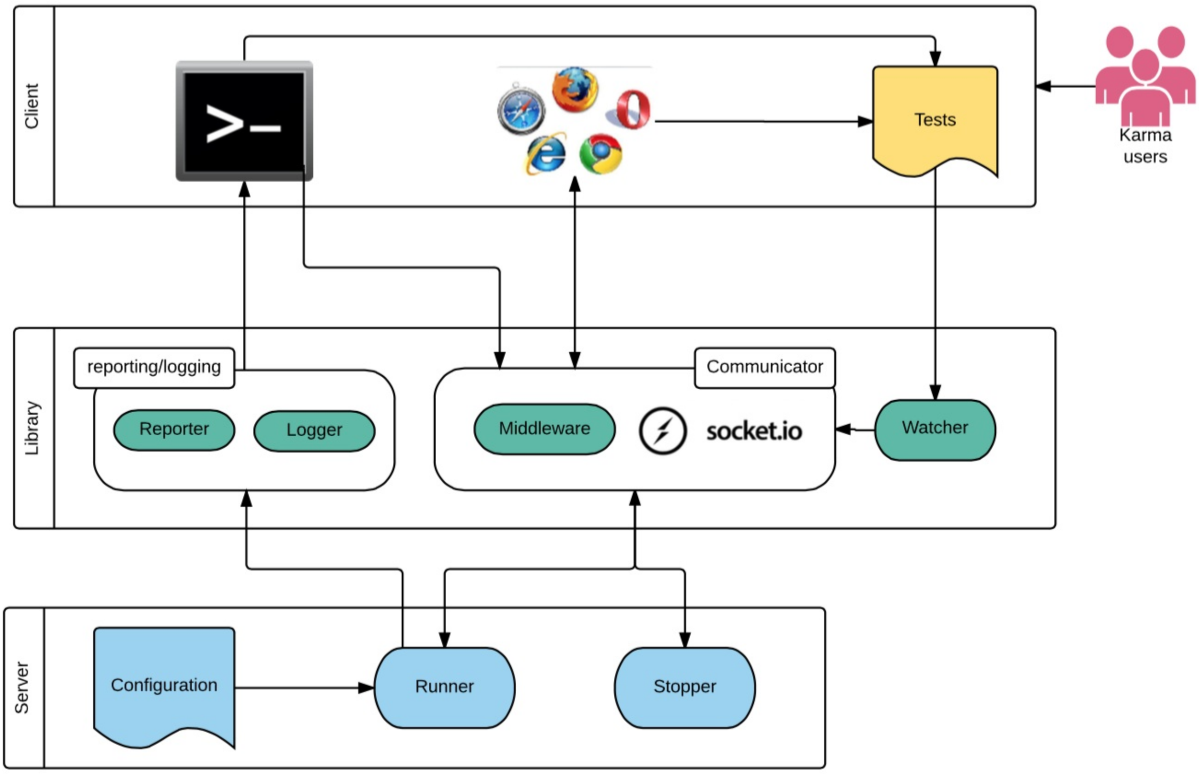


Figure 3. *The Internal Environment*

External Environment

In order to explain and depict the system's interactions with the external environment, a context model is made. This model is used to explain what the system does, and does not do. It presents an overall picture of the system’s interactions with the outside world, and summarizes the roles and responsibilities of the participants in these interactions.

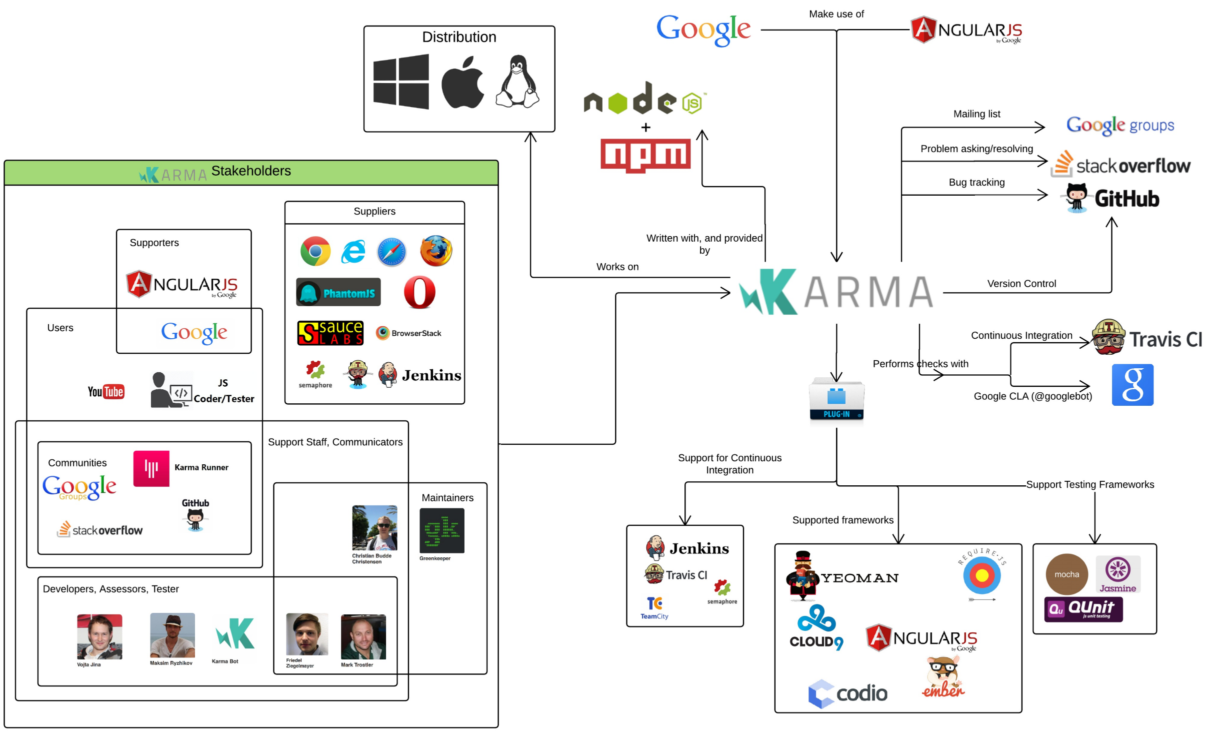


Figure 4. *Context Model*

Impact on the Environment

Now that is known what the overall environment consists of, the impact on the environment will be discussed. The impact of the Karma's deployment on the aforementioned environment is addressed in this part. According to *Rozanski and Woods*, addressing the impact on the environment includes:

1. Systems that contain dependencies on Karma and may require
   - functional changes,
   - interface changes,
   - performance improvements,
   - security improvements;
2. Systems that will be decommissioned as a result of Karma's deployment;
3. Any data that will be migrated into Karma.

The continuance of this subsection will discuss each of the aforementioned concerns regarding the impact on the environment.

- **Dependent Systems**

  Karma is a test runner which integrates testing frameworks. Especially for the purpose of continuous integration. Since Karma is often used as an end-to-end test runner (running independently of the rest of the framework), other systems do not depend on the way Karma works. Other systems integrate Karma by running the Karma program and edit Karma's configuration options, in order to have matching interfaces. As Karma provides this functionality, other systems are dependent on whether or not Karma will implement the configuration interface. This is the case for all levels of dependency: *functional*, *interface*, *performance* and *security*.

- **Decommissioned Systems**

  There are several possible scenarios where systems will be decommissioned. This might not always be the case, and when systems will be decommissioned due to a fault in Karma, it will be noticeable. Karma will not decommission many systems, because of Karma's end-to-end characteristics. Karma is well known for its enormous amount of plugins and its flexibility. So, it aims to prevent the decommissioning of systems. In case systems get decommissioned, this might have an (in)direct effect on the plugins and other functionalities within Karma.

- **Data Migration**

  With regard to data migration, Karma takes in JavaScript tests created by the end user. Since this type of data is often of a universal template, Karma does not influence, nor have a large impact on the way data is migrated into Karma.

Functionalities

Karma's environment shows a large number of actors. Moreover, the previous section discussed several systems who are dependent on Karma or may even be decommissioned once Karma stops working. In order to understand more clearly on which processes are relied on by other systems, the functionalities of Karma will be discussed. The main goal of a system can in general be derived from the sum of its functionalities. The main goal for Karma is *"to bring a productive testing environment to developers"* [2]. The central functional capability provided by Karma is, therefore, the ability to run different (types of) tests. There exist five main functionalities which are identified, and elaborated on, in the following table:

Functionality	Description
Running tests	The ability to run tests on a various amount of different browsers and operating system platforms.
Remote control	Controlling the entire workflow either through the command line or the IDE.
Testing framework integration	Integration with testing frameworks, which can be used to generate test descriptions. Some testing frameworks are: Jasmine, Mocha or QUnit.
CI integration	Integration with Continuous Integration servers like Jenkins, Travis or Semaphore.
Easy debugging	Possibility to directly debug, in an easy way, from the IDE (like WebStorm) or Google Chrome.

Table 2. *Functionalities and their corresponding description.*

Design Philosophy

The stakeholders identified in the environment analysis should be taken in mind and continuously kept satisfied during the endless development cycle of Karma. Each type of stakeholder has different requirements in terms of Karma's software architecture. The stakeholders influence the design philosophy, which is translated in terms of design qualities and their appliance to the system. These design qualities and their appliance are elaborated on in the following table:

Design Quality	Appliance
Separation of Concerns	Karma has been made in such a way that elements are separated from each other. In this way, elements can be grouped together depending on their task. The result is that it is much easier to support Karma.
Cohesion	The Karma system has a high cohesion available. The strength of the relationship between pieces of functionality within a given module is high. Within the middleware part of Karma, functionalities are grouped together since they all contribute to a single, well-defined task.
Coupling	Karma provides the ability to create plugins for different tasks. This is done to ensure loose coupling between certain system components. Since the system is loosely coupled, it provides the ability to easier build, support and enhance the system.
Functional flexibility	With the functional capabilities in mind, it is difficult for Karma to have a high functional flexibility. It is the core of the system and the additional parts have been developed as plugins. Changing the functional elements would result in breaking (multiple) other components.

Table 3. *Design qualities and their appliances.*

Functional Structure Model

Figure 5 depicts the functional structure model of the Karma system. Different functionalities are illustrated as end nodes in the diagram, pointed to by an arrow. These functionalities include, for example, the functionality of *CI Integration* or *Test Results*. Different interfaces are required in order to reach a certain functionality. These interfaces are either internal, provided by the Karma system, or external, provided by means of dependencies or separate installation. One could, for example, clearly see that in order to gain CI Integration functionality, one requires the server-interface from within the Karma system as well as other external testing frameworks like Jasmine or Cucumber. These frameworks, are not provided from within the Karma system and are thus said to be external interfaces.

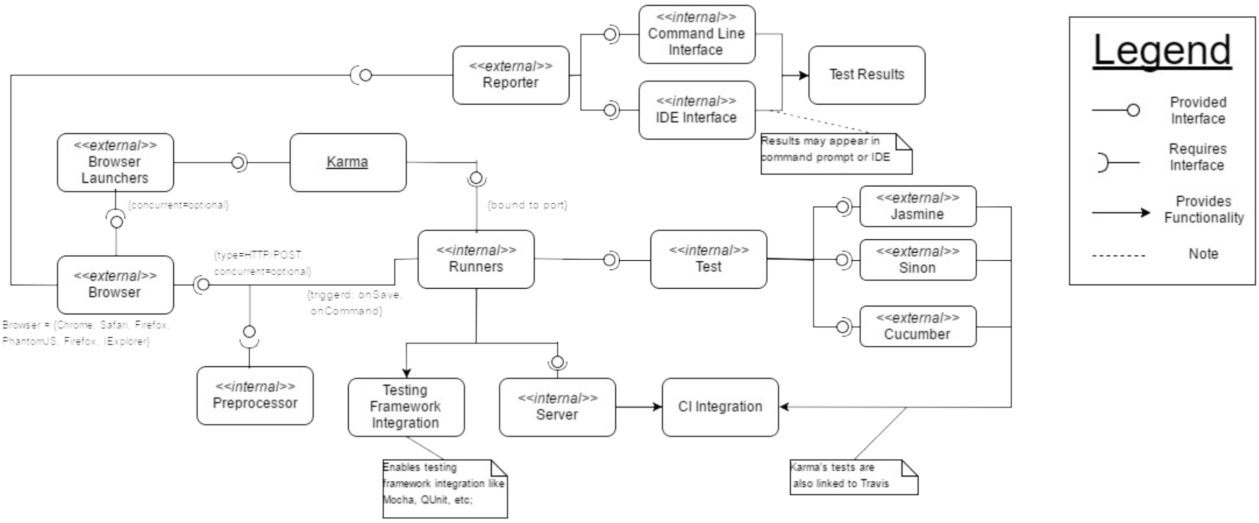

<<internal>> Command Line Interface
<<external>> Reporter
Test Results
<<internal>> IDE Interface
Results may appear in command prompt or IDE
<<external>> Browser Launchers
Karma
{concurrent=optional}
{bound to port}
<<external>> Jasmine
<<external>> Browser
{type=HTTP/POST, concurrent=optional}
<<internal>> Runners
<<internal>> Test
<<external>> Sinon
{triggerd: onSave, onCommand}
Browser = {Chrome, Safari, Firefox, PhantomJS, Firefox, IExplorer}
<<external>> Cucumber
<<internal>> Preprocessor
Testing Framework Integration
<<internal>> Server
CI Integration
Enables testing framework integration like Mocha, QUnit, etc;
Karma's tests are also linked to Travis
Legend
Provided Interface
Requires Interface
Provides Functionality
Note

Figure 5. *Functional Structure Model*

Variability

Features are needed to aid the existence of the main functionalities of Karma. These features do not only aid the functionalities, but may also affect the variability and flexibility of the system. In order to identify the variability of the system, these features are required to be identified at first. With these features, it is possible to deduct the flexibility of the system. The variability aspects described in this section, all aim to keep the technical debt as low as possible.

Main Features

The main advantage of using Karma as a test runner is to gain instant testing feedback from test code, while in the meanwhile, freedom on customizing Karma is also allowed to fit the user's preferences. The most important features are listed below in Table 4.

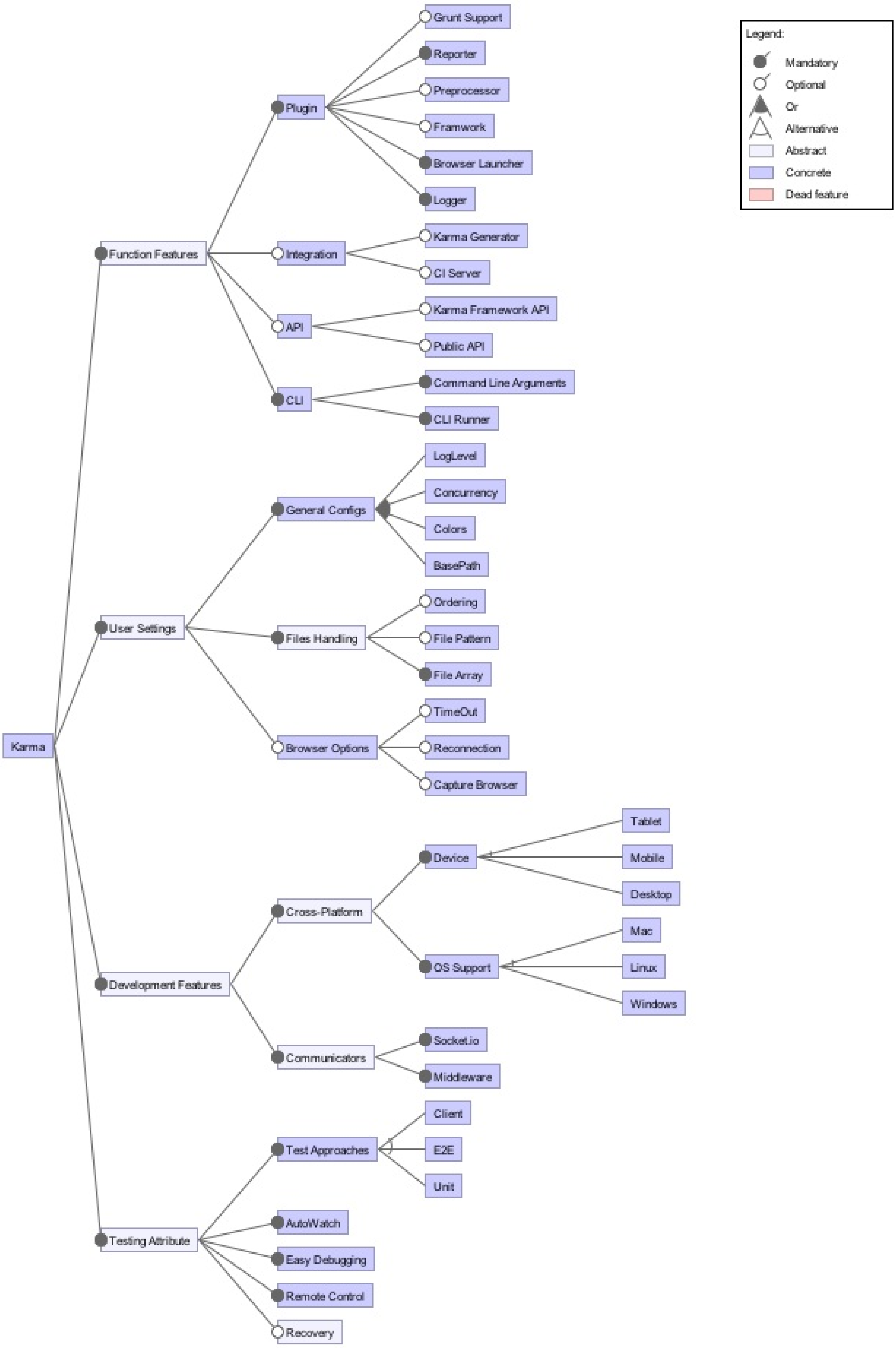
Feature	Description
Cross-Platform	Karma is platform independent and hence supports different OS (Windows, Linux, MacOS)and devices (desktop, mobile, tablets).
Plugin	Many external products (browser-launchers, testing frameworks, preprocessors, reporter, etc.) can be integrated with Karma via plugins, which maximizes Karma's abilities.
Configuration	Configuration enables customize Karma's running behaviors. In this file, users specify wanted plugins, customize browser options, decide how to handle testing files and basic settings like color of logging information etc.
CLI	Karma can be run directly through the command line and command line arguments have ability to overridden settings predefined in the configuration file.
Public API	Karma API provides users the ability to call Karma programmatically from different Node modules and in the meanwhile enable users to write customized plugins to connect existing testing libraries.
Easy Debugging	Debugging directly from Chrome or your IDE via WebStorm. After each modification users can see the test output directly in the IDE and do not need to switch to the terminal.
Auto Watch	Karma triggers tests automatically once a file is saved w.r.t corresponding changes. It enables automatic debugging and provides real-time test results to users.
Test Approaches	Client, unit and e2e tests approaches help users focus on certain aspects they want when performing tests.

Table 4. *Features with their corresponding description.*

Relationships

Sometimes one feature can not arise without the existence of another feature or features. This generates feature dependency or even feature interdependency. A dependency in Karma is the browser-launcher, which relies on the type of operating system and corresponding devices. The easy debugging attribute depends a lot on the Chrome-launchers and IDE-plugins. Remote control is realized in the existence of CLI and other IDE(s). Recovery feature appears partially due to the reconnection and timeout features of browser setting. Other than these, much more of such situations are detectable in the case of Karma.

Considering the respectable amount of features, the most important features are classified into four categories according to their properties with relation to their relationships. The classification consists of: *development features*, *function features*, *user settings* and *testing* *attributes*. Most of the identified features and their relationships are illustrated in the feature model below.



"Browser Launcher" ⇒ "OS Support" ∧ Device

"CLI Runner" ⇒ Desktop

"Browser Options" ⇒ "Browser Launcher"

AutoWatch ⇒ "User Settings"

"Test Approaches" ⇒ Framwork ∧ "Browser Launcher" ∧ "Files Handling"

"General Configs" ⇒ Logger ∨ Reporter ∨ "Browser Launcher"

"Remote Control" ⇒ "Command Line Arguments" ∧ "CLI Runner"

Recovery ⇒ TimeOut ∨ Reconnection

"Files Handling" ⇒ BasePath

"Testing Attribute" ⇒ "Development Features" ∧ "Function Features" ∧ "User Settings"

Figure 6. *Feature Model*

Strategy

In order to flexibly provide these features to the large number of individual preferences originating from different kinds of developers, Karma is highly configurable. Karma also integrates with popular continuous integration packages and has excellent plugin support. Karma's success is mainly due to its variability, configurability and thus its flexibility. To be able to keep this up, a strategy is of importance to apply to the system to keep it successful. Below, three aspects are summed up which aid this variability, configurability and provide the flexibility which aid this strategy.

- **Configuration File**

  Users of Karma are able to modify the configuration file to fit their needs. This configurability ranges from changing a port number to the plugins used within the system. There are mainly five sections of the configuration file that influence the configurability of Karma the most: *testing framework*, *preprocessors*, *browsers* and *files handling*. The configuration file exists with default values set which are changeable. Karma extracts this information and applies it throughout the system, without any further need of modification.

- **Plugin**

  As aforementioned, Karma provides the ability to use plugins in the system. This provides variability to the users of the system. The wanted plugins are set inside the configuration file and loaded at the start of Karma. Depending on the functionality of the plugins, they are loaded at the appropriate parts of the system. A simple example is the use of a different browser. This is loaded as a plugin, and in case it is found, that browser is also run during the testing process.

  Note: It is advised to perform an `npm install` to install plugins which are not present yet.

- **Design Patterns**

  To provide users with more variability, Karma provides some files which can be specialized by means of inheritance. In the Karma project, these files, or variable functions, start with the word `base`. An example of such a file that can be used to apply inheritance on, is the *BaseReporter*. The use of inheritance in the project makes it possible for users to specify in more depth what they need. In case a certain file/module does not provide the needed functionality, this makes it easier for them to change. Especially the messages, output by Karma, can be changed easily in this manner.

Development View

Now that a higher overview of Karma is gained, this section dives deeper into the development process. Complex systems such as Karma are often built in several modules. These modules are organized in a specific way. This section addresses the aspects which support the software development process and starts first of all by identifying all different modules and lastly provides a discussion on the existence of code standards.

Modules

Karma consists of a large number of different source files, which for maintainability have been logically organized into several different modules. The following modules have been identified in Table 5.

Module	Description	File amount	Size (bytes)
Runner	Makes it possible to run Karma from the command line	1	50
Client	Provides the client shown to the user when Karma starts	7	11k
Documentation	Documentation of the Karma project	29	80k
Library	Overall library	49	142k
Library - General	Provides general functionality for the server, loading plugins etc.	28	100k
Library - Initialization	Questions regarding for the installation of Karma (the configuration)	3	8k
Library - Launcher	Contains parent launchers, these can be extended for custom launchers	4	9k
Library - Middleware	Creates the communication for the Karma client. Serving files and proper setup of the HTML-pages.	7	20k
Library - Reporter	Reporters for the Karma client. Both for tests and the debug view.	7	8k
Node Modules	Contains all the modules used by Karma. E.g. test frameworks, browser launchers, server sockets etc.	N/A	N/A
Static	Static HTML-files shown in the browser.	4	11k
Tasks	Provides initialization for the development environment and ability to perform *grunt tests*	2	2k
Tests	Contains the tests for Karma	77	250k
Tests - Client	Client tests and the karma configurations.	6	23k
Tests - Unit	Unit tests for specific parts of the system.	33	21k
Tests - E2E	End-to-end tests for expected behavior and data integrity.	36	205k

Table 5. *Modules of Karma.*

All modules contribute to the end-to-end functionality and, therefore, require interconnection. The following figure, Figure 7, shows Karma's Module Structure Model. Within this figure, the interaction between these modules is depicted.

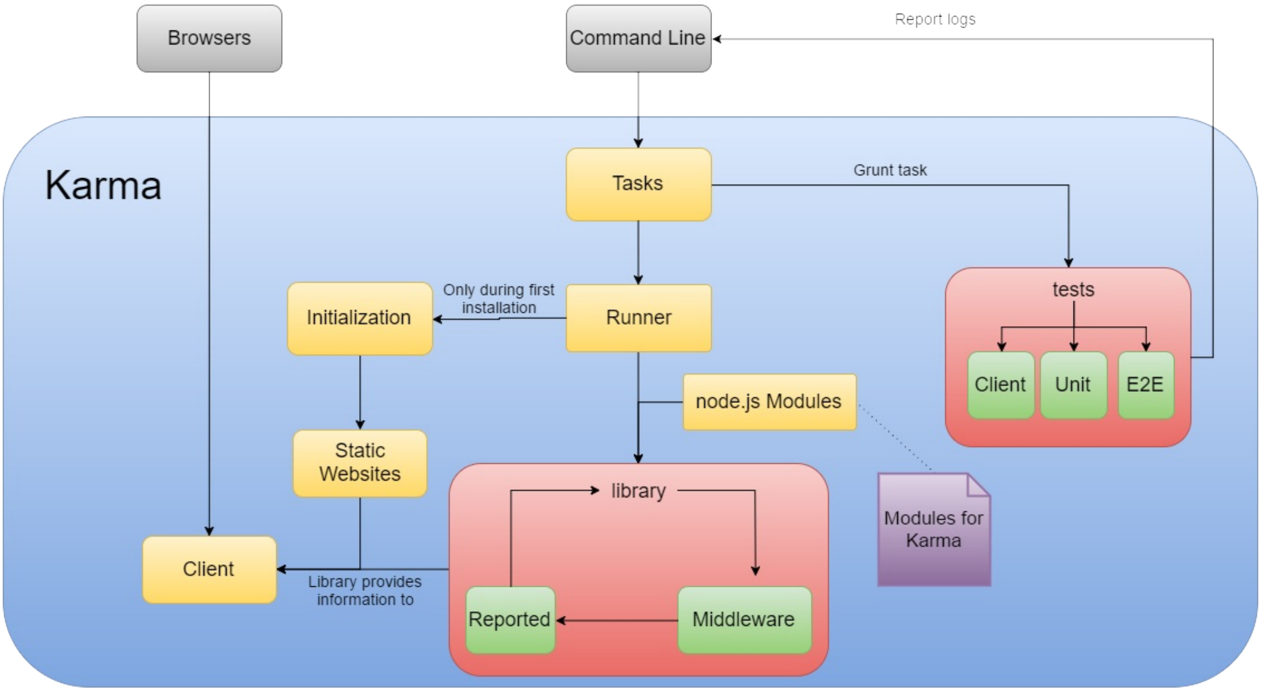


Figure 7. *Module Structure Model*

Code Standards

With the aforementioned modules, what kind of standards does Karma apply to its development process? Most software systems, and especially systems that rely on the open source community, are developed by multiple people rather than individual persons. In order to collaborate effectively, it is best practice to adhere to standardized key aspects of design and testing.

Standardization of Design

Most of the functionality of the Karma system is implemented using three layers, as seen in Figure 3. Karma's main functionality is running tests as Karma is called a *test runner*. The tests are specifically run on the server, whereas the results are all reported to one main client, which prompts the user with the test results. The bridge is provided by the library layer, which is in essence the backbone of Karma. So, this layered approach is a standardization of their design.

Standardization of Testing

Karma uses three general approaches to test Karma's internal functionality and integrity. The test approaches are: *client*, *unit* and *e2e* (end-to-end). Certain frameworks have been used, which each enforce some standardization of test implementation. Table 6 summarizes these test components.

Approach	Framework	Explanation
Unit	Jasmine and Sinon	Small tests, testing low-level method functionality.
Client	Jasmine and Sinon	Larger tests, testing higher-level system functionality.
E2E	Cucumber	Full feature tests.

Table 6. *Standardization of Testing in Karma.*

Technical Debt

With knowledge of the development view, the existence of technical debt can be analyzed. Software developers sometimes derogate quality in order to ship changes fast. This always leads to technical debt, because the future effort is required in order to recover the initial quality. As can be deducted from the section about variability, this all helps to reduce the technical debt. The following section describes how the technical debt can be managed especially for Karma.

Management of Technical Debt

In order to manage the technical debt, while also reassuring code quality, Karma's issues and pull requests on GitHub [3, 4] have been studied. By assessing this, each task has a brief description of the technical change to be made, why this technical change is important for the project and in which part of the code the technical change has to be performed (if known).

According to Frank Buschmann's, To Pay or Not to Pay Technical Debt [5], one could either *repay the debt*, *converse the debt* or *pay the interest*.

Which option to choose should be decided by the stakeholders, since we need to see the project in business perspective and consider many aspects. In the case of Karma, the third option is adopted in general: **pay the interest**. Karma relies on the developers in the open-source community to solve issues and provide pull requests. However, not all the bugs in issues are solved by developers, some of the issues are actually solved by users and some of the issues are just left there unsolved.

Throughout the issues of Karma, the general choice of implementing new features is done by is by providing a clean and smart solution. This takes longer, but makes changes easier in the future [6]. An example of such a situation is issue #1976. To solve this issue, it is "quite intricate and will involve a good amount of changes inside the code base" [7].

So, to manage the technical debt, Karma focuses on keeping the variability within the system and take their time to implement new features to have a good solution. There may exist many issues and PRs for Karma at the moment, but this is done to have the best solution for the issue. This will keep the technical debt as low as possible.

Performance Perspective

With all the previous sections in mind, this section aims to analyze Karma's performance in terms of possible workload and how Karma handles increased processing volumes. To be more precise, this section includes performance analysis on the most valuable components of the system. Below, five elements are summed up which encapsulate concerns regarding the performance of a system [1]. Next, Karma is put to a test to deduct and visualize the response time to create a performance model.

- **Response Time**

  One of the main issues for a JavaScript developer is the time of testing. This often takes several minutes which slows down the developer in his process. This results in a long and ineffective process. One of the goals of Karma is to tackle this problem. Thus, fast-speed has become one of the main properties of Karma as it provides instant feedback to users. This is further discussed and depicted in the following subsection.

- **Throughput**

  Throughput is defined as the amount of workload the system is capable of handling in a unit time period. Throughput and response time have a complex interrelationship in most systems and so it is in Karma. In general, the shorter transaction processing time in Karma, the higher the throughput Karma can achieve. This is an important aspect as it influences the speed of the amount of tests capable of being run.

- **Scalability**

  Scalability is the ability of Karma to handle the increasing and short burst of workload. This may be due to an increase in the number of files or an increase in the complexity of tasks. Since Karma watches the files and performs tests the moment they have been changed/saved, this makes the workload not so high and aids the scalability of the system. Furthermore, Karma is able to run browsers concurrently, spreading the workload. This all aids the scalability aspect of the system.

- **Predictability**

In a system, predictability is also of importance. Regardless of when a certain task is executed, the task should execute in a very similar time. This is the case for Karma. However, during the first initial run of Karma, it seems that little is cached, resulting in the fact that it takes a longer time than subsequent runs. This, in overall, makes the system predictable.

- **Hardware Resource Requirement**

  With much flexibility, Karma works on multiple platforms (Windows, Linux, Mac etc.). The tests can even be captured on several hardware components, such as the desktop, mobile, tablets etc. This has an effect on the throughput and the response time of the system. Nevertheless, Karma aims to keep this cross-platform functionality going and makes sure to have it working on the most known platforms.

Performance Model

The model below shows the most valuable work components within Karma. In order to elaborate on the performance more precisely, tests have been carried out to get the response time data on different functional parts. From Figure 8 can be seen that most steps in Karma have an 'instant' response time, which are usually within 2 seconds. As aforementioned, the first run of Karma takes more time, probably because nothing has been cached in memory yet. Furthermore, the reporter may take up to 5 seconds due to the length of reports. The overall performance, in terms of response time, can all be seen in the figure.

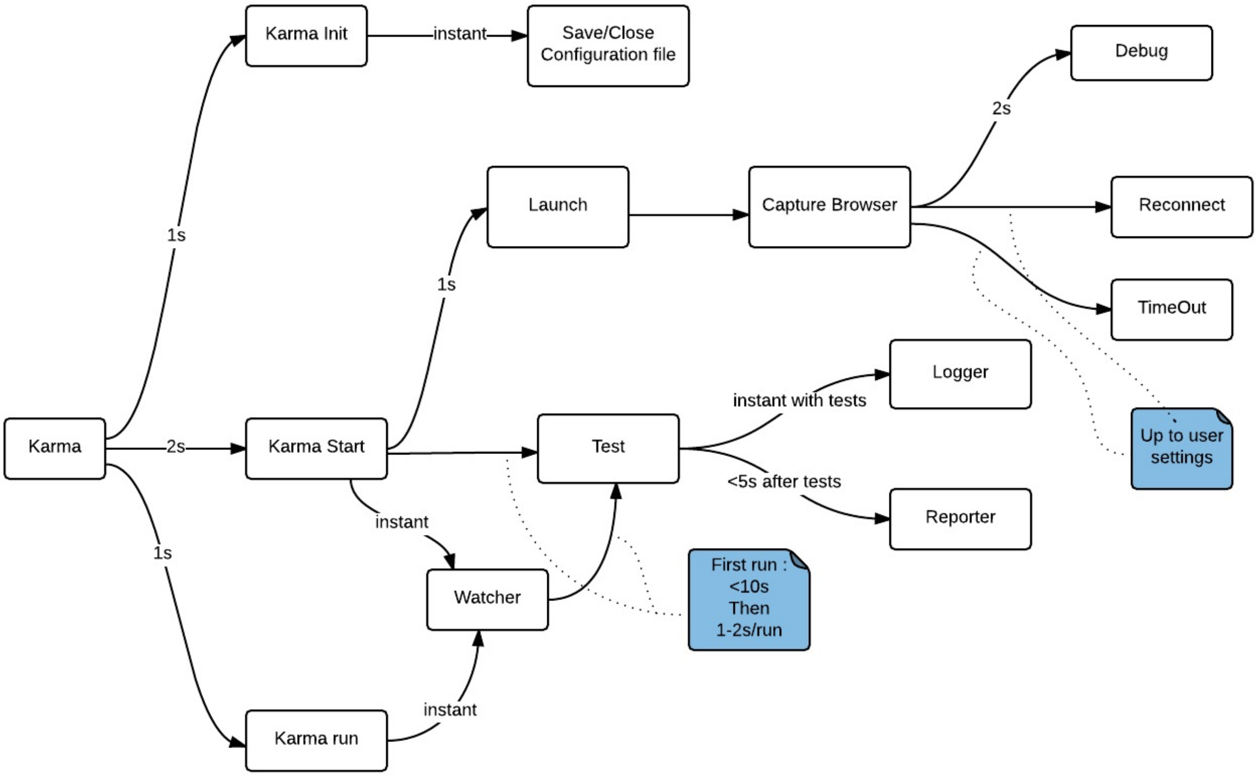


Figure 8. *Performance Model*

Conclusion

The main goal for Karma is to bring a productive testing environment to JavaScript developers. The environment being one where they don't have to set up loads of configurations, but rather a place where developers can just write the code and get instant feedback from their tests.

The number of stakeholders in Karma is quite big, but there are only a few developers that frequently work on karma. The feature of Karma is abundant due to its rich number of configuration files, this character offers the user various choices and makes Karma one of the most popular JS test runners.

Karma is the preferred test runner for projects written with AngularJS and is well on its way to larger acceptance within the JavaScript community. Its plugin architecture makes it easily adaptable to other test suites and reporters, all of which add value to the core of Karma. In agile or continuous integration environments, Karma shines as an indispensable tool to development teams, providing an easy and reliable way to modify existing code and craft new code.

In conclusion, with its amount of functionalities, variability, low technical debt and the great performance, it is a great tool to achieve the goal it was made for. To bring a productive testing environment to JavaScript developers.

References

1. Rozanski, Nick, and Eóin Woods. Software systems architecture: working with stakeholders using viewpoints and perspectives. Addison-Wesley, 2012.
2. The official Karma website: http://karma-runner.github.io/
3. Karma's issue page: https://github.com/karma-runner/karma/issues
4. Karma's pull request page: https://github.com/karma-runner/karma/pulls
5. Buschmann, Frank. "To pay or not to pay technical debt." Software, IEEE 28.6 (2011): 29-31.
6. Fowler, M. (2016). Martinfowler.com. Retrieved 27 March, 2016, from http://martinfowler.com/bliki/TechnicalDebt.html
7. Ziegelmayer, F. (2016). Retrieved 27 March, 2016 from https://github.com/karma-runner/karma/issues/1976#issuecomment-194349020

Mopidy – An extensible music server

By Yuzhu Yan, Lina He, Yuchen Huang, Chang Yan

Delft University of Technology

Abstract

Mopidy is an extensible music server written in Python. In this research, the software architecture of Mopidy is analyzed through the stakeholders and various viewpoints including the context viewpoint, developer viewpoint and deployment viewpoint. Sequentially, the perspectives of variability and usability are provided to address the quality properties, which will have a positive impact on the improvement of Mopidy. From the analysis of viewpoints and perspectives, it can be seen that Mopidy, as an open source API, still need to be enhanced.

Contents

Introduction

Mopidy is an extensible music server. It allows users to connect their favorite MPD or web client to the multiple cloud services like Spotify, SoundCloud and Google Play Music. The users can add new music source easily and control the music from any phone, tablet, or computer.

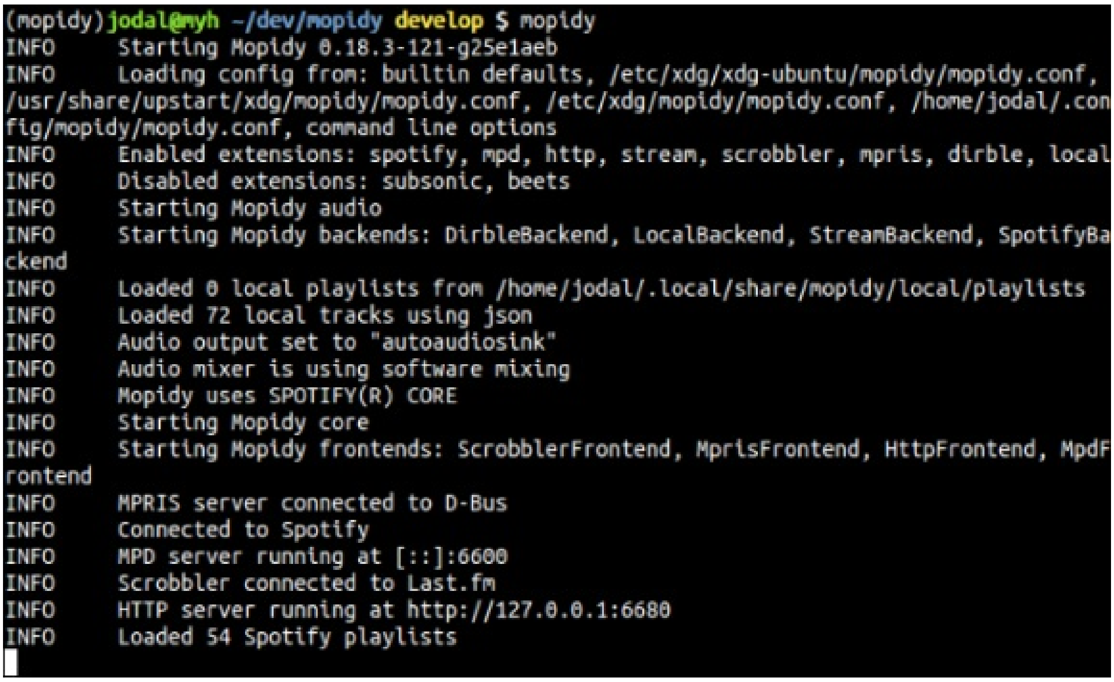
Team-Mopidy, from the Delft Students on Software Architecture group (DESOSA 2016), has analyzed the architecture of Mopidy based on the theory about software architecture over the past few weeks. From these analysis, we want to do some contribution to the development of Mopidy. This chapter is divided into various viewpoints and perspectives about Mopidy. First of all, there is a brief introduction about Mopidy from general and specific understanding.

General Understanding

No matter frontends or backends, so far, there are more than 10 extensions for Mopidy. With the help of extensions, music from cloud service, like Spotify, SoundCloud, and Google Play Music can be easily added. Through Vanilla Mopidy, music from your local disk and radio streams could also be enjoyed.

Cloud services of Mopidy

Mopidy is just a server

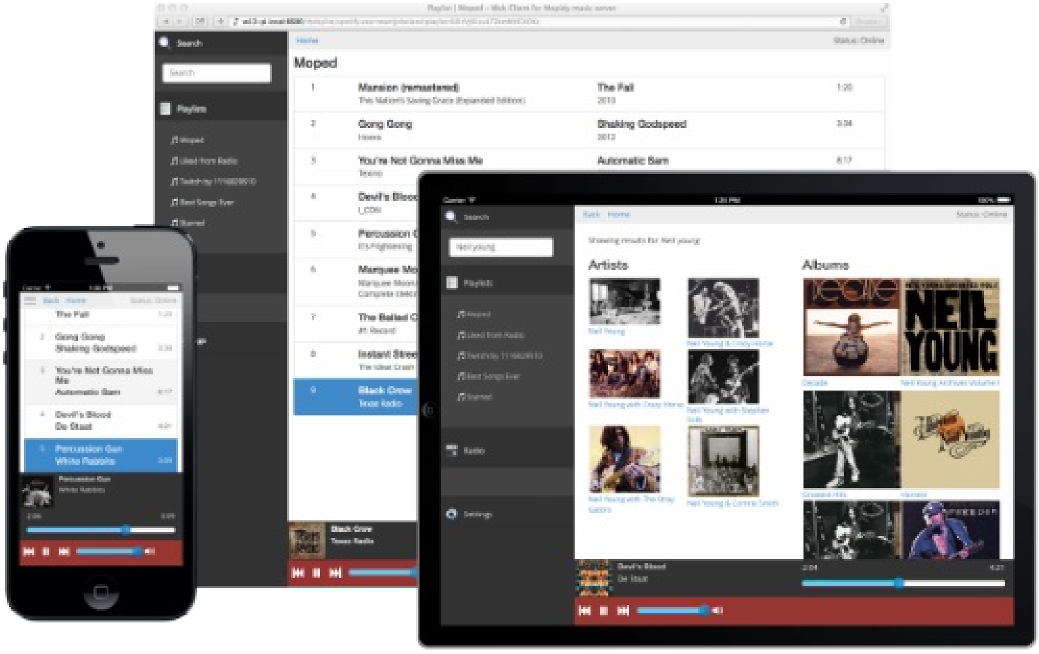


```
(mopidy)jodal@myh ~/dev/mopidy develop $ mopidy
INFO     Starting Mopidy 0.18.3-121-g25e1aeb
INFO     Loading config from: builtin defaults, /etc/xdg/xdg-ubuntu/mopidy/mopidy.conf,
/usr/share/upstart/xdg/mopidy/mopidy.conf, /etc/xdg/mopidy/mopidy.conf, /home/jodal/.con
fig/mopidy/mopidy.conf, command line options
INFO     Enabled extensions: spotify, mpd, http, stream, scrobbler, mpris, dirble, local
INFO     Disabled extensions: subsonic, beets
INFO     Starting Mopidy audio
INFO     Starting Mopidy backends: DirbleBackend, LocalBackend, StreamBackend, SpotifyBa
ckend
INFO     Loaded 0 local playlists from /home/jodal/.local/share/mopidy/local/playlists
INFO     Loaded 72 local tracks using json
INFO     Audio output set to "autoaudiosink"
INFO     Audio mixer is using software mixing
INFO     Mopidy uses SPOTIFY(R) CORE
INFO     Starting Mopidy core
INFO     Starting Mopidy frontends: ScrobblerFrontend, MprisFrontend, HttpFrontend, MpdF
rontend
INFO     MPRIS server connected to D-Bus
INFO     Connected to Spotify
INFO     MPD server running at [::]:6600
INFO     Scrobbler connected to Last.fm
INFO     HTTP server running at http://127.0.0.1:6680
INFO     Loaded 54 Spotify playlists
```

The Interface of Mopidy

As an application written in Python, Mopidy not only have network connectivity, audio output, but also support multiple operating systems and device, from linux, Mac OS X to Raspberry Pi. Out of box, Mopidy takes a role of server for MPD and HTTP. Additional frontends for controlling Mopidy can be installed from extensions.

Everybody uses their favourite client

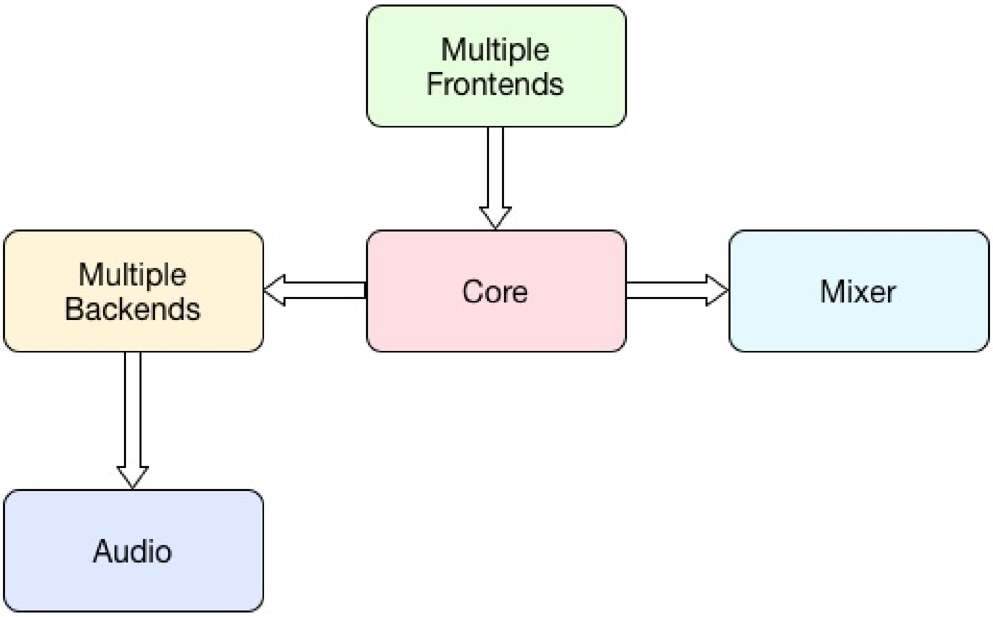


Clients of Mopidy

You are not just a beneficiary, but also a contributor. You and people around you could connect their favorite MPD or web client to the Mopidy server to search music and manage playlist together. By making use of a browser of MPD client, which is available for all popular operating systems, control music form any phone, tablet or computer is not a imagination any more. [1]

Specific Understanding

The main function of Mopidy is to make multiple frontends capable of using multiple backends. The overall architecture of Mopidy as shown below is organized based on the frontends and backends.

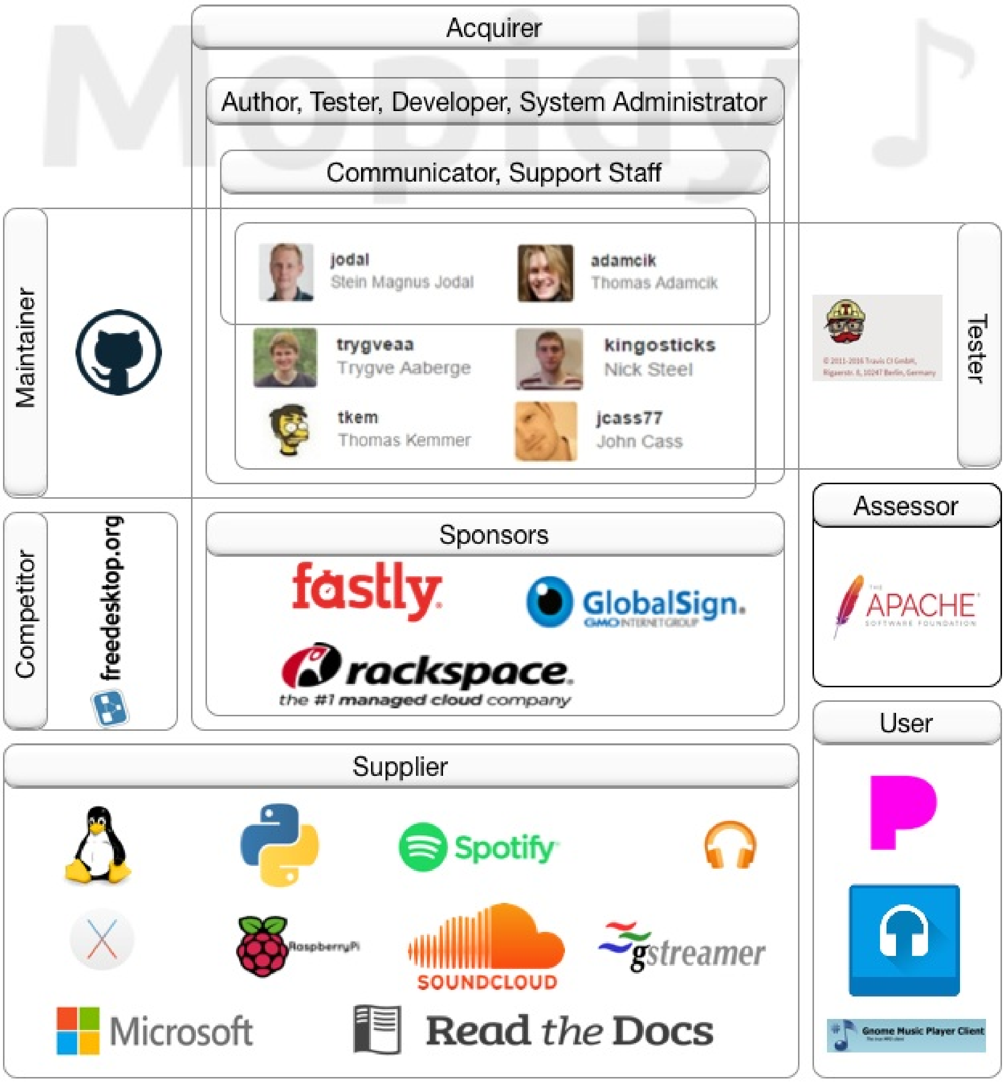


Architecture of Mopidy

The multiple frontends present the resource from Mopidy to user. They can implement a server for protocols such as HTTP, MPD as well as MPRIS. The core contains several controllers for different functionalities. The core actor gets the requests from the frontends and calls out to one or more backends. The responses from backends will be combined into a single response and sent to the requesting frontend. In this way, the multiple backends are able to work as one. Another function of the core is keeping track of the tracklist. In the backends, there are a set of providers having different functionalities to provide various music sources. Additionally, the audio is built around GStreamer for backends to play audio. Mixer is used for the volume control.[1]

Stakeholder Analysis

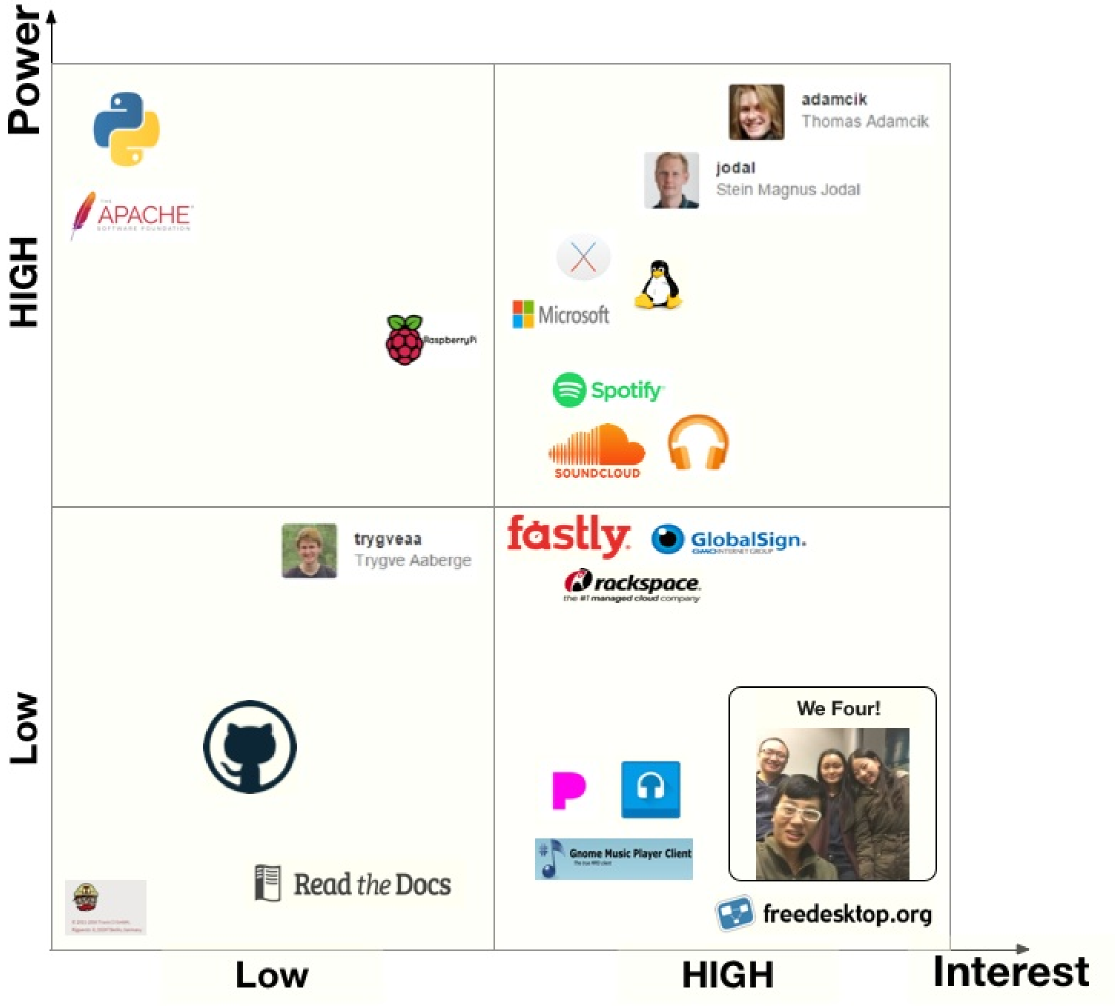
Types of Stakeholders



Stakeholders of Mopidy

Mopidy is developed, maintained and tested by Mopidy Group mainly composed by 8 people. They made great contributions and gave life to Mopidy. They are not only the authors of Mopidy, but also the acquirers and administrators. Some of them @Jodal and @Adamcik also play the role of the system communicators and supporting staffs. Mopidy got sponsored by Fastly, GlobalSign and Rackspace. Also Mopidy is assessed by Apache which supports a number of open-source software projects. Mopidy has many suppliers such as raspberry pi, Spotify and SoundCloud. Furthermore, Mpd clients and Mpd graphical clients are its main users.

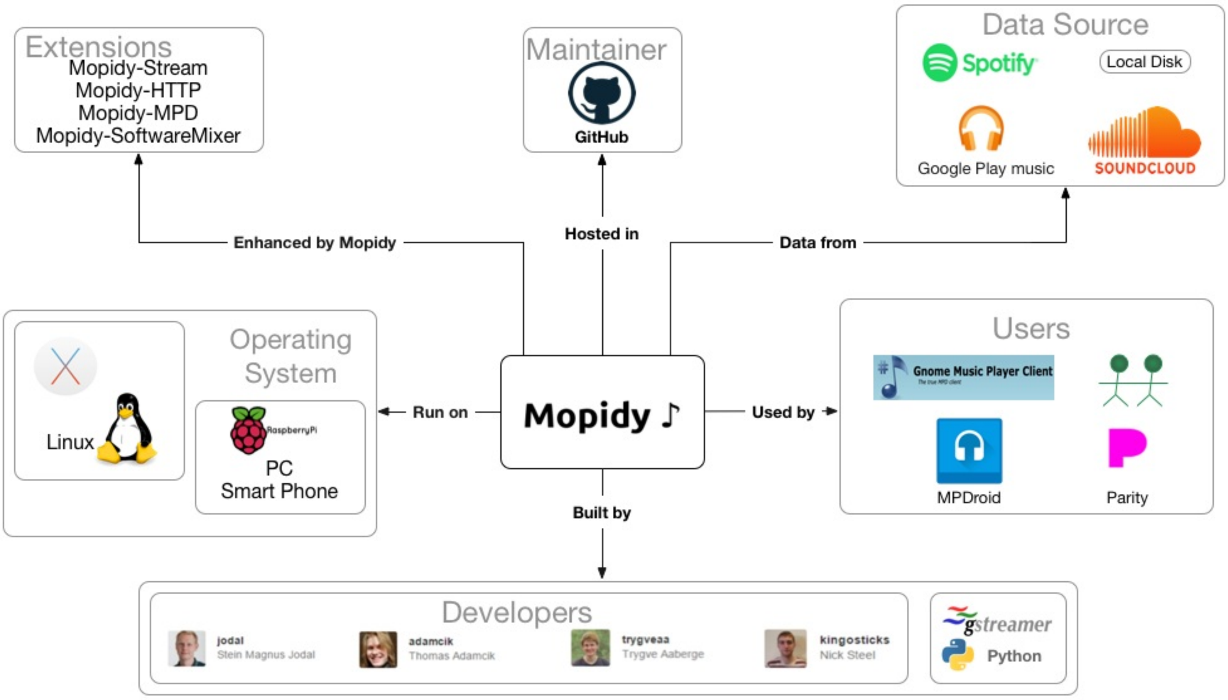
Power-Interest Grid



Power-Interest Grid of Mopidy

Given the power and interest in Mopidy, the stakeholders are prioritized as shown in the Power-Interest grid. In this figure, the high power and interest stakeholders like Jodal and Adamcik are placed on the top right, while stakeholders with both lower power and interest including some of the testers, maintainers and suppliers are in the bottom left corner. Other places in this grid show stakeholders with either high power or high interest such as our four team members who has high interest but low power in Mopidy.

Context Viewpoint



Context Viewpoint of Mopidy

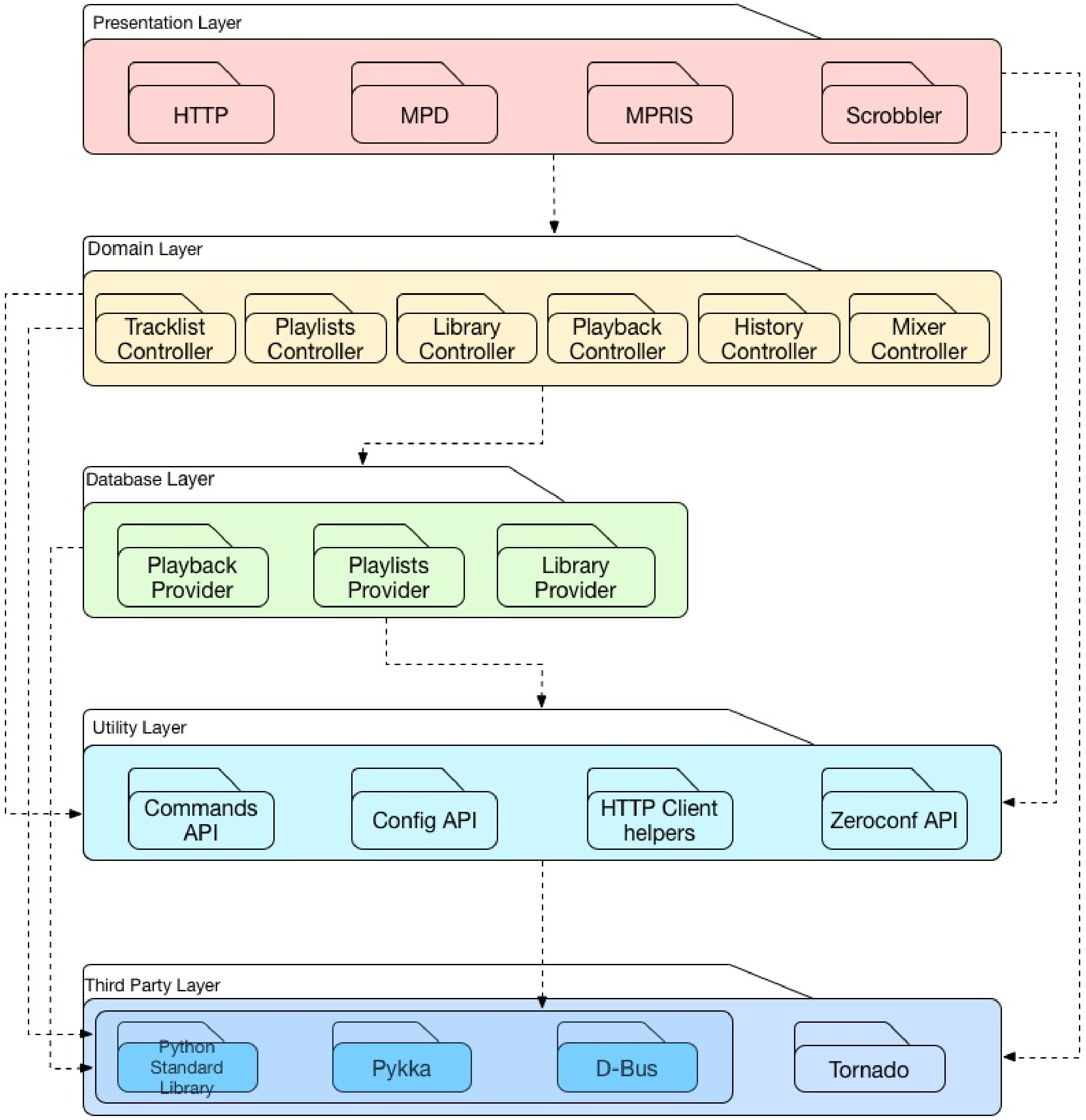
Mopidy is a completely open source project and GitHub is the main platform in which the Mopidy is developed. Like any project hosted in GitHub, the developers come from everywhere as long as they are interested in Mopidy and are willing to contribute. Developers who has made great contribution to the project may be invited in the Mopidy member team. Those people are the core developers of the Mopidy. Mopidy originally supports local disk and Spotify, other stream source and additional functions can be achieved by some third party extensions. As an open source project, users can easily download the source code and build the program from GitHub. Not only personal users, there also some software, such as Parity, use the code from Mopidy project with their own UI. Mopidy can run on PC with OS X and Linux, the support for Windows and smart phones is also on the way. Competitors like PulseAudio from freedesktop.org can also provide the similar function.

Developer Viewpoint

When developing the system, the Mopidy developers have to consider a lot, such as the code structure, dependencies, configurations, constraints and design standards, to ensure the integrity and feasibility of the system. In this part, there is an analysis about the development architecture of Mopidy during the whole development process.

Module Organization

Before writing the source code, the developers will consider the element of Mopidy to form an extensible music server. In this part, the module organization is introduced to achieve the main functionalities of Mopidy.



UML Module Structure of Mopidy

The module structure of Mopidy is presented as a UML package diagram. In this diagram, there are five layers from presentation layer to third party layer.

- Presentation Layer: The presentation layer is the frontends directly providing the resource for user and can implement server for protocols such as HTTP, MPD as well as MPRIS.
- Domain Layer: The domain layer contains several controllers for different functionalities like keeping track of the tracklist and etc.

- Database Layer: In the database layer, there are a set of providers having different functionalities to provide various music sources.
- Utility Layer: The utility layer contains many modules used by other layers.
- Third party layer: The third party layer has several platforms which are imported to achieve the a set of functionalities.

The before-mentioned five layers depend on each other to some extent. The layers connected by arrows mean that the dependency of modules in these layers are allowed, for example, the modules in presentation layers are allowed to depend on the utility layer, domain layer and third party layer.

Common Design Model

Define a set of design constraints that apply when designing the system's software elements helps increase the system's overall technical coherence and makes it easier to understand, operate, and maintain. It can reduce risk and duplication of effort by identifying standard approaches to be used when solving certain types of problems as well. The definition of common design model can be specialized into three parts: common processing, standard design approaches and common softwares.

The Common Processing

Standardizing how the system logs messages and handles configuration parameters can significantly simplify its administration. Common processing provides a standard approach across all system elements that can benefit a lot and directly contributes to the overall technical coherence of the system.

Message Logging

- Use *mopidy –v* or *mopidy –vv* or *mopidy –vvv* to print debug log. All three options will give you debug level output from Mopidy and extensions, while *–vv* and *–vvv* will give you more log output from their dependencies as well.
- The console log always based on log level, defaults to true
- The *config file* that overrides all logging config values. The *console format* used for informational logging. The *debug format* used for debug logging. See the Python logging docs for details on the format.
- The file to dump debug log data to when Mopidy is run with the *mopidy –save-debug-log option*. It will save the log equivalent with *–vvv* to the file *mopidy.log* in the directory you ran the command form.
- During development or debugging, the *loglevels* config section can be used to change the log level for specific parts of Mopidy. But each key in the config section must match

the name of a logger. The value is the log level to use for that logger, one of *debug*, *info*, *warning*, *error or critical*

- During development or debugging, the *logcolors* config section can be used to change the log color for specific parts of Mopidy. Each key in the config section should match the name of a logger. The value is the color to use for that logger, one of *black*, *red* , *green*, *yellow*, *blue*, *magenta*, *cyan*, *white*.

The Configuration

For the initialization, Mopidy has a lot of config values you can tweak, but you only need to change a few to get up and running. When you have created the configuration file, open it in a text editor, and add the config values you want to change. You can see the config by typing *mopidy config*. All extensions bring additional configuration values with their own defaults. You can check the core config, logging config, audio config and proxy config and change them as you want. All configuration sources are merged together to show the effective document. For termination and restart of operation, Mopidy has its commands on mopidy and commands on extensions.

The Standard Design Approaches

Standardization of Testing

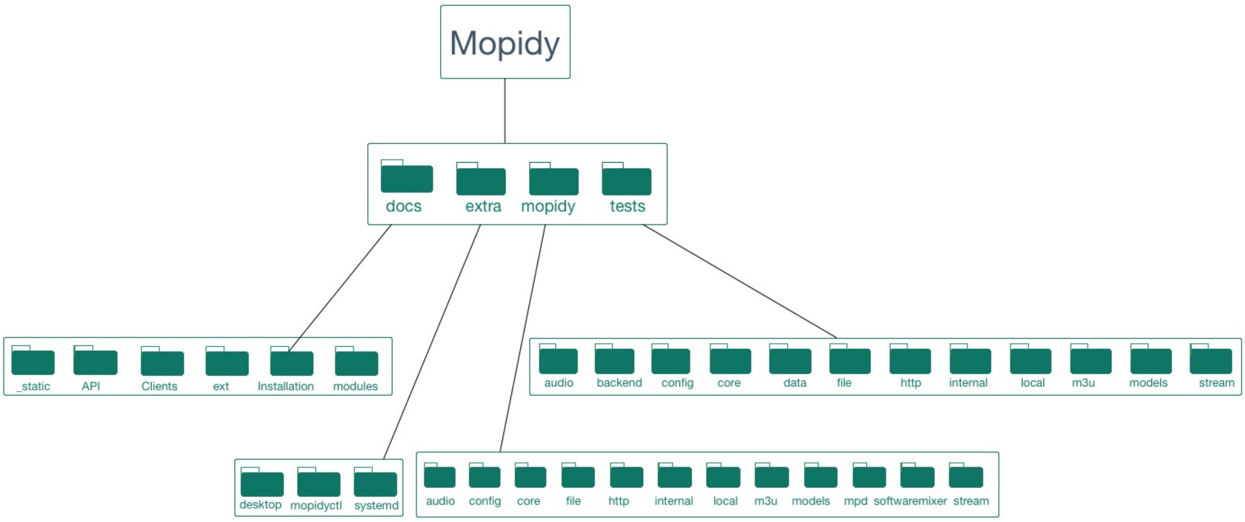
Mopidy has quite good test coverage, and all new code going into Mopidy are advised to come with tests. Mopidy team provides many effective and practical test tools. “tox”, for example, a command that could run all tests, including all branches and pull requests. This is the ultimate test command before pushing to Github. Pytest as a test runner suffices people who want just want to run a test in a single directory to save time. And with the help of the pytest-cov plugin, you can even check ten slowest tests with aim of speeding up. All the codes modifications are welcomed written in the suggesting code style. As a convenience, Mopidy team also provides a directory named tests on Mopidy Github root directory. Developers could see detailed test in this file.

Testing Tools

Testing tools which are available for Mopidy includes: Travis CI, Tox, pytest, flake8.

Codeline Organization

As an open source project, it is highly important to keep the source code in order so that other developers can easily understand and make contribution to it. Codeline organization is considered to be the way to store source code in a directory structure, managing it via configuration, building and testing the system regularly. Testing and configuration has been discussed in the previous model, this part is mainly focused on the source code structure and the build and release regulation.



Codeline Organization of Mopidy

Under the root directory, Mopidy is divided into four subdirectories, docs, extra, mopidy and tests respectively. The docs folder contains the reStructuredText files for manuals and other important information. The manuals are also separated to different folders according to their content. The mopidy subdirectory contains the source code for the actual implementation of the Mopidy server. As we have discussed before, Mopidy is well organized by dividing its functionalities into different modules. The structure of the mopidy subdirectory is also based on the modulation of the whole system's design. Each module has its own directory to store its source code. This keeps all the source code in order and makes it easier for developers to analyse each module. The tests folder is also parted to subfolders. Each of them only tests its counterpart under the mopidy folder.

As Python is an interpreted programming language, Mopidy is not necessary need to be built before it gets released. Developers, as well as users, only need to type the command mopidy, then the system will be automatically built and run.

In order to reduce the project's dependency on key individuals and to automate the process, release procedures are well defined:

1. Update changelog and commit it.
2. Bump the version number and update the test case.
3. Merge the release branch into maser.
4. Install/upgrade tools used for package.
5. Build package and test it manually in a new virtual environment.
6. Tag the release.
7. Push to GitHub.
8. Upload the previously built and tested sdist and bdist_wheel packages to PyPI.

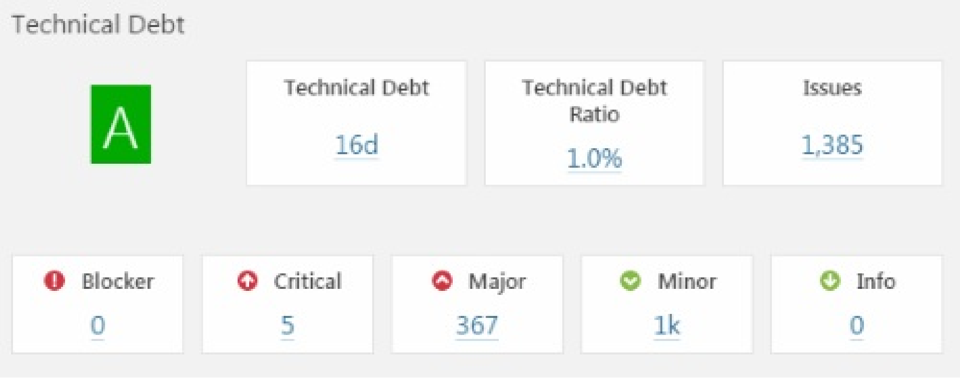
9. Merge master back into develop and push the branch to GitHub.
10. Make sure the new tag is built by Read the Docs, and that the latest version shows the newly released version.
11. Spread the word through the topic on #mopidy on IRC, @mopidy on Twitter, and on the mailing list.
12. Update the Debian package.

Technical Debt

During the process of software development, there will be more or less some work which needs to be completed before a particular job but is neglected by the developers. These neglected work will cause plenty of technical debt which will block the development of the project. Actually, there are many causes of technical debt like the business pressures and lack of understanding. We believe that some technical debt may be caused by Mopidy developers, so in this section we use a tool to identify the technical debt in Mopidy.[4]

Technical debt involved in Mopidy

We examined Mopidy code files by using SonarQube, an open source software used for quality management. Whenever a code violates basic rule of the programming language, SnoerQube propose an issue. By using this software, we received the following results.



Technical Debt

A

Technical Debt 16d

Technical Debt Ratio 1.0%

Issues 1,385

Blocker 0

Critical 5

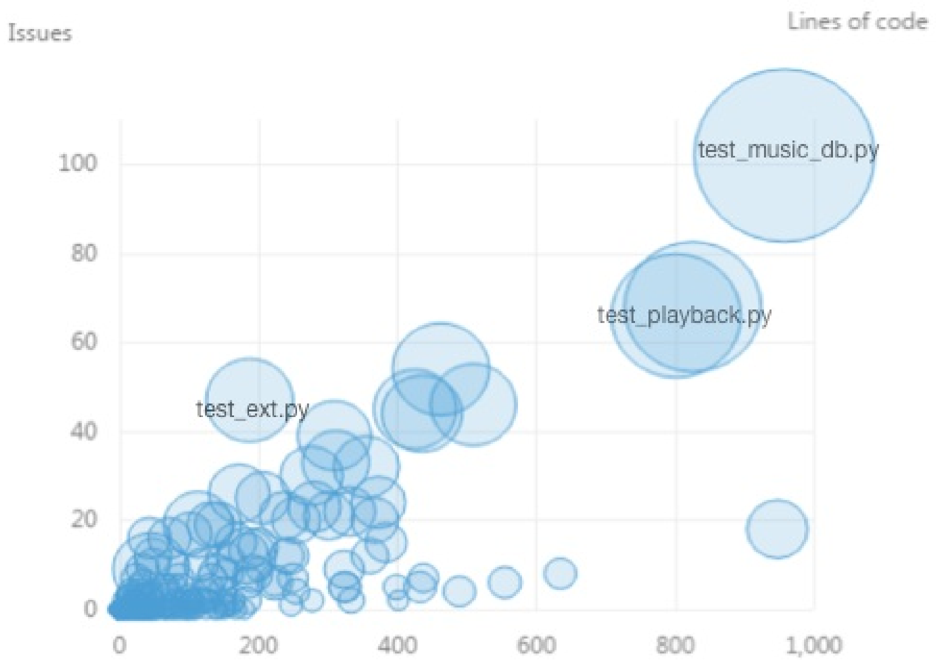
Major 367

Minor 1k

Info 0

Technical debt illustration

From above consequence, we can see there is no blocker in Mopidy codes. Five critical issues are showed, which are caused by rename aiming for avoiding misunderstanding. Class methods lead to major issues. Furthermore, minor issues are raised up due to rename methods. Luckily, technical debt ratio is showed as 1.0%, which means Mopidy is well-organized and maintained.



Technical debt of files

One circle represents a file of Mopidy. The size of circle represents the number of issues in a file, which means, the more issues exist, the larger circle is. The horizontal axis stands for lines of code in a file. So we can conclude that the top right circle should contain highest number of issues and largest amount of codes. This circle is the file of test_music_db.py, which includes 57 lines of code, 102 issues and 25 hours technical debt.

Deployment Viewpoint

According to the definition in book "Software Systems Architecture", the definition of deployment viewpoint describes the environment into which the system will be deployed, including the dependencies the system has on its runtime environment.

So far, Mopidy is supported by the following operating systems, Debian/Ubuntu, Arch Linux, OS X, Raspberry Pi. For Debian/Ubuntu, the packages of Mopidy are built for:

- Debian wheezy (oldstable), which also works for Raspbian wheezy and Ubuntu 12.04 LTS.
- Debian jessie (stable), which also works for Raspbian jessie and Ubuntu 14.04 LTS and newer.

The packages are available for multiple CPU architectures: i386, amd64, armel, and armhf. For Arch Linux, package size of Mopidy 2.0.0-2 is 3.0MB, the installed size is 9.6MB. Mopidy runs on all versions of Raspberry Pi. However, note that Raspberry Pi 2 B’s CPU is approximately six times as powerful as Raspberry Pi 1 and Raspberry Pi Zero, so Mopidy will be more joyful to use on a Raspberry Pi 2. Furthermore, in order to ensure the developers know what software will be available for them and to make the system administrators know what needs to be installed and managed on each piece of hardware, third-Party Software requirements must be clearly identified. The dependencies of Mopidy varies from different operating systems. All of these information is showed in the following table.

Table1 The operation systems and third-party software

Operating Sytems	Third-Party Software
Debian/Ubuntu	GStreamer, Gst-plugins-ugly, Gst-python2
Arch Linux	gst-plugins-good, gst-plugins-ugly, gst-python2, gstreamer, python2, python2-pykka>=1.1, python2-requests, python2-setuptools, python2-tornado>=2.3, mopidy-mopify (virtual) (optional) - Web client with Spotify-inspired interface, mopidy-scrobbler (virtual) (optional) - Last.FM scrobbler, mopidy-soundcloud (virtual) (optional) - Soundcloud backend, mopidy-spotify (virtual) (optional) - Spotify backend, python2-sphinx (make)
OS X	Xcode, XQuartz, Homebrew, Python
Raspberry Pi	GStreamer, Gst-plugins-ugly, Gst-python2

Variability Perspective

As an extensible music server written in Python, developers of Mopidy devoted all their attention to attract different clients and people who are using different kinds of devices, in a range of Linux, Mac OS X, Raspberry Pi, normal people, music fans etc. As a result, developers must take variability as an important element into account when they are designing and developing this API.

Feature Identification

The Mopidy, as a music server, can be divided into some modules and each module has plenty of features for different types of stakeholders. In this part, the features of Mopidy are identified.

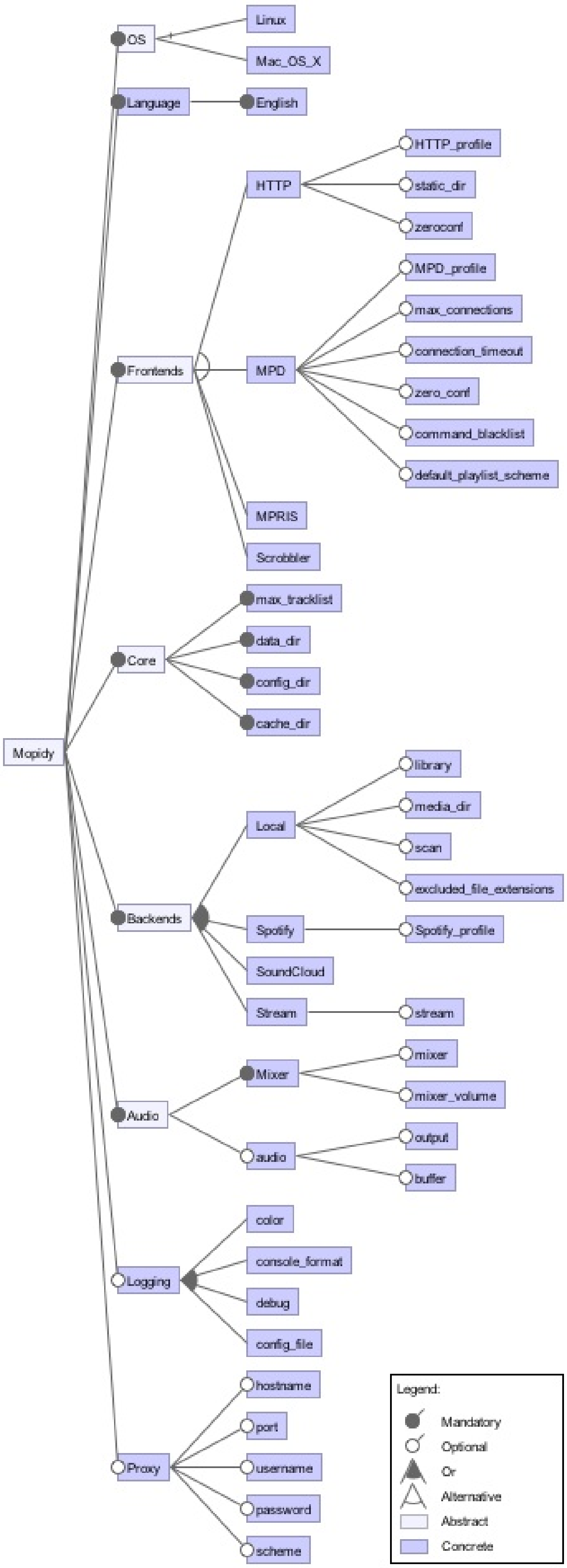
- Frontends：Mopidy provides many API for different types of frontends like HTTP, MPD, MPRIS and Scrobbler. The user can set up the API configuration about the specific frontend to make Mopidy suitable for this frontend. For example, The Mopidy-HTTP can be used for a web client to control Mopidy through HTTP and WebSockets. The features in frontends are mainly related to the Mopidy user such as MPDroid and Partify.

- Core：The core contains several controllers for different functionalities. The core actor gets the requests from the frontends and calls out to one or more backends. It has many configurations for user to control the paths and the tracklist length.
- Backends：In the backend module, there are a set of suppliers like local, Spotify, stream and Soundcloud providing various music sources. These music sources can connect Mopidy and user can configure many parameters.
- Audio: The audio is the interface around GStreamer for some specifc cases. The default playback provider is used when the backend is able to simply set the URI of the resource. But sometimes for example the raw audio data is delivered outside of GStreamer, the audio is needed to execute some additional functions. Furthermore, The audio is able to control the volume of default mixer.
- Logging：Logging is used for user to inspect the running status of Mopidy. There are many parameters can be set up to change the attribute of console log, debug results and get the configuration file.
- Proxy：It is used for users or suppliers on locked down networks. So far not all parts of Mopidy and Mopidy extensions respect the proxy server configuration.
- Language and OS：The language feature for Mopidy is English. It is relevant to all stakeholers. And Mopidy is compatible for two operation systems(OS): Linux and MAC OS X. This feature is bound up with the suppliers.

Feature Relationships

Product line is defined by its features. Their relationship and a specific product is identified by a subset of features, this process is called a feature selection. Generally, there could be many possible limitations when the developers are defining feature selection. The constraint of feature selection named feature dependency. As a result, we define a feature selection is valid if and only if it fulfills all feature dependencies. After identifying basic feature dependencies, subsets of features can be decided, from which we can derive the feature model which can be regarded as a formalism to describe features and their constraints[3].

For Mopidy, we did domain analysis and derive the following domains: operating system(OS), language, frontends, core, backends, audio, logging and proxy. We derive the following *Feature variability model of Mopidy* and constructed it by FeatureIDE.



Feature Variability Model of Mopidy

Notes: In order to make the model concise, well-organized and accommodate more information, we use summarized feature instead of a class of features.

- feature *debug* indicates debug_file and debug_format
- feature *HTTP__ _profile* indicates hostname and port
- feature *scan* indicates scan_format, scan_flush, and scan_follow_symlinks

Binding Time

Like we introduced above, Mopidy can run on different systems and devices. This really offer great convenience for users, however, conflicts, lack of hardware and many other issues emerge. So how Mopidy cope with incompatible problems is a critical point for Mopidy development. Luckily, so far, these variabilities are solved during Build Time.

Build Time

A feature during Build Time means it can be regarded as a static binding. This is because it occurs before run time and remains unchange throughout program execution. Examples of this are OS and frontends for Mopidy. An operating system should be determined before installing Mopidy, and can never be changed during the run time unless you use it on another device or install Mopidy again. It is similar to frontends. Compared to OS, you have more options of frontends and it is more flexible because it is unecessary to uninstall Mopidy if you want an alternative frontend. The only thing you need to do is adding some configuration in the configuration file of Mopidy like the code fragment1. But it have to be noticed that this feature can only be initialised during build time.

```
[http]
enbaled=true
hostname = 127.0.0.1
port = 6680
static_dir =
zeroconf = Mopidy HTTP server on $hostname
```

Code fragment 1: Shows the variable features during build time.

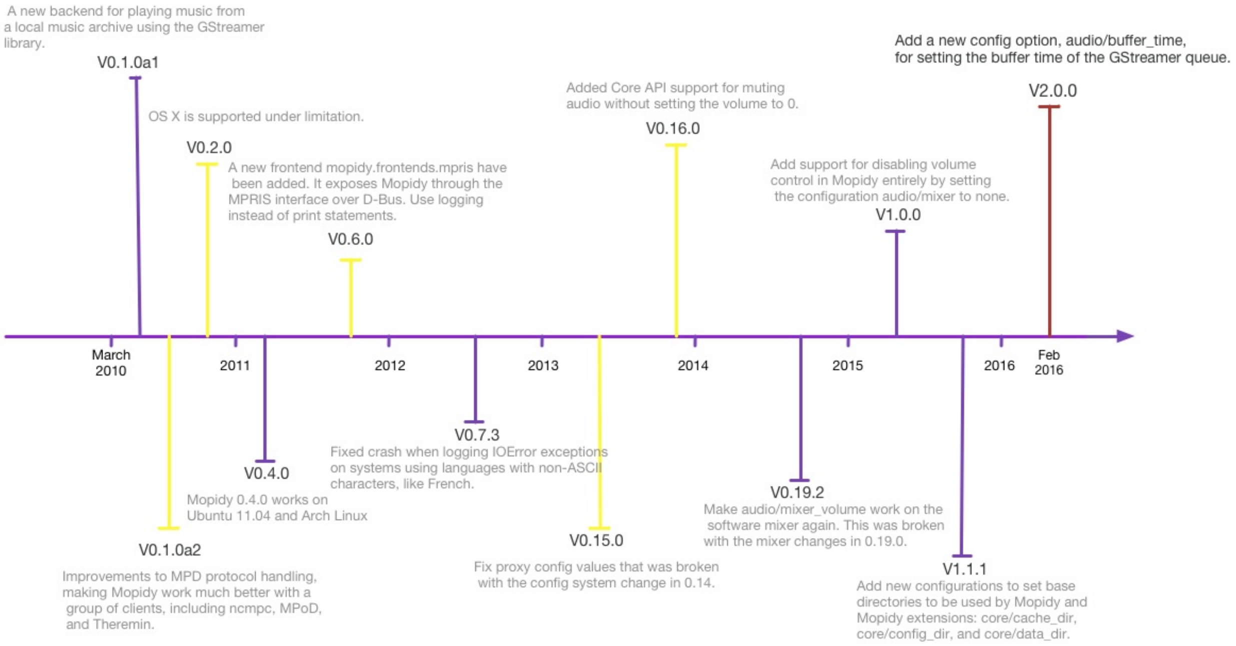
It can be seen that *http/enabled* decides if HTTP extension be used or not. And the hostname listens on IPv4/IPv6 loopback interfaces. And http/port shows which TCP port the HTTP server should listen to. The developers of Mopidy deprecated *http/static__ _dir* and said this will be removed in a future version of Mopidy. For the last item, *http/zeroconf*, means the name of the HTTP service when published through Zeroconf. The variabiles *$hostname* and *$port* can be used in the name.

Similar to frontends, Mopidy also provides multi functions. Through modifying configurations in configuration file of Mopidy, user can combine different functions and get a 'Unique Mopidy'. And users can also set different configurations in logging file to select partial logging information. All above mentioned features only work after restarting Mopidy. So we define these as build time variabilities.

It is interesting that the relationship of cost and the variability is like, a trade off. Developing the simplest system probably costs less at start because the system is simpler and quicker to deliver, but each later change is likely to cost more because developers have no existing mechanism for implementing the change. How to balance the cost and variable features is obviously the most important issue which need to be concerned about during the development of Mopidy.

Evolution History

The evolution history of the variability mechanism and configurable features is summarized based on the changelog provided by the Mopify document. The version history of Mopidy is showed in the following picture.[1]



V0.1.0a1: A new backend for playing music from a local music archive using the GStreamer library.

V0.1.0a2: Improvements to MPD protocol handling, making Mopidy work much better with a group of clients, including ncmpc, MPoD, and Theremin.

V0.2.0: OS X is supported under limitation.

V0.4.0: Mopidy 0.4.0 works on Ubuntu 11.04 and Arch Linux

V0.6.0: A new frontend mopidy.frontends.mpris have been added. It exposes Mopidy through the MPRIS interface over D-Bus. Use logging instead of print statements.

V0.7.3: Fixed crash when logging IOError exceptions on systems using languages with non-ASCII characters, like French.

V0.15.0: Fix proxy config values that was broken with the config system change in 0.14.

V0.16.0: Added Core API support for muting audio without setting the volume to 0.

V0.19.2: Make audio/mixer_volume work on the software mixer again. This was broken with the mixer changes in 0.19.0.

V1.0.0: Add support for disabling volume control in Mopidy entirely by setting the configuration audio/mixer to none.

V1.1.1: Add new configurations to set base directories to be used by Mopidy and Mopidy extensions: core/cache_dir, core/config_dir, and core/data_dir.

V2.0.0: Add a new config option, audio/buffer_time, for setting the buffer time of the GStreamer queue.

Mopidy version history

Pull requests and issues show a power influence when deal with the variability of the Mopidy project. For example, in issue #1409 and pull request #1442, a bug was caused by the fixed buffer size and now the buffer size is configurable. Another example is, in pull request #901, the compatibility issues of Python 3 were almost fixed, users now can build Mopidy by Python 3 instead of the old version. This is also a response to issue #779.

Usability Perspective

The usability perspective mainly focus on the end users of the system and any others who interact with the system directly or indirectly [2]. The usability cannot be ignored for the implementation and development of a system. It significantly affects the usage experience of the users and other interactors. In Mopidy, the users, maintainers and suppliers are relevant to the usability. Taken these stakeholders into consideration, the usability of Mopidy is analyzed in this part for the sake of efficiency of the interactions.

The first step for the usability perspective is to identify touch points. Touch points are the places for people to interact with the system. Mopidy is a server to connect multiple backends and the frontend. So users interact with Mopidy only through the terminal in OS X or Linux as FigureThe Interface of Mopidy shown.

The users can operate Mopidy in the terminal by many commands to achieve install, start, stop the system and etc. They can also change the config values directly in the configuration file named mopidy.conf to set up Mopidy like hostname and other parameters of each modules. And users can run mopidy or run Mopidy as a service.

Mopidy is an internal system, which is tightly to be controlled and managed. It can only be operated through commands on the terminal. So the users should know how to operate the OS (Operating System) in the terminal. It is not easy for the majority of people because people are accustomed to use graphical user interface like Windows. This will have a significantly negative impact on the usability of Mopidy.

Conclusion

Mopidy is a server to allow you to play music from local disk, Spotify, SoundCloud and other sources. This project started at 2010 and the development of Mopidy is completely based on GitHub. With more and more people contribute to Mopidy, now it has evolved to the version v2.0.0. Although the active evolution makes Mopidy increasingly powerful, it is still far from completion. A lot of functions which are originally intended to be added in are still empty. And also, the compatibility becomes a severe problem. Deprecated APIs cannot be removed due to the extensions which are still using them. All these problems give us opportunities and challenges to provide our contribution to Mopidy.

References

1. Stein Magnus Jodal, Johannes Knutsen, Thomas Adamcik. Mopidy documentation

2. Nick Rozanski, Eoin Woods. Software Systems Architecture

3. Sven Apel, Don Batory, Christian Kästner, Gunter Saake. Feature-Oriented Software Product Lines

4. D. Sculley, G. Holt, D. Golovin, E. Davydov, T. Phillips, D. Ebner, V. Chaudhary, M. Young, and J.-F. Crespo. "Hidden technical debt in machine learning systems". In Neural Information Processing Systems (NIPS). 2015.

Neo4j - A Graph Database

Kangliang Chen, **Manoj Krishnaraj**, **and Tom Peeters**

Delft University of Technology

Abstract

This chapter gives a brief overview of Neo4j's architecture by focusing on three fundamental concepts: stakeholders, viewpoints, and perspectives as defined in the book by Rozanski and Woods[1]. The stakeholder analysis section details the types of influencer's who have an impact the Neo4j's architecture. The context view explains the interactions between Neo4j and its environment; the development view and deployment views describe the software development process and the runtime environment respectively. Following this, the evolution, variability, performance & scaling perspectives of Neo4j are analyzed. Through these multiple viewpoints and perspectives, the software architecture of Neo4j can be easily discerned.

Table of Contents

Introduction

In the early 2000s the scalability of relational databases hit a ceiling, further increases in performance became difficult. These performance problems led to the conceptualization of graph database by its creator, Emil Eifrem, on an airplane flight. As in a graph structure, graph databases use nodes and edges to represent and store data. Semantic queries can be easily performed as the data stored in the nodes are interconnected with related nodes via edges. Neo4j is based on the above principles of a graph database. Neo4j is named after the Latin word for new (neo) and is partly inspired by the character, *Neo*, in the movie *Matrix*. The first version of graph database was deployed in 2003 and its source was made public in 2007. The important milestones in Neo4j's history are depicted in Figure 1.

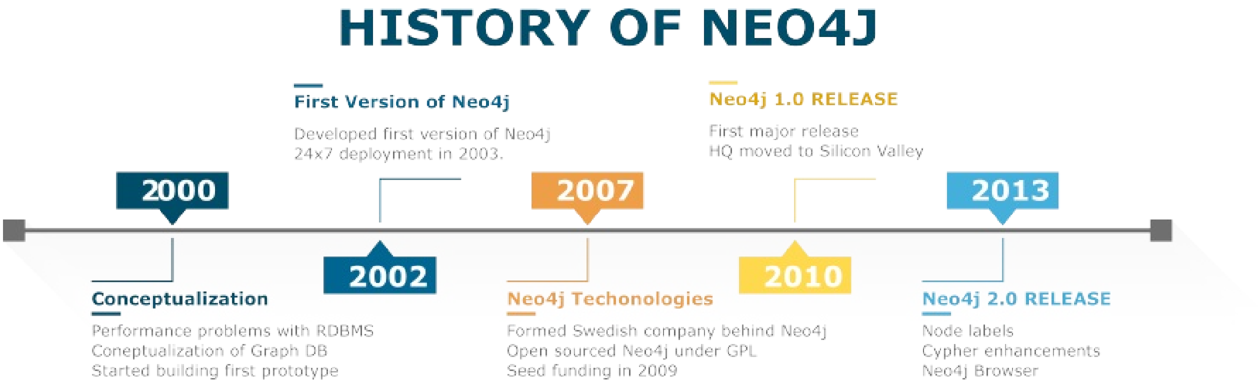


Figure 1: Timeline of important events in Neo4j's history

Neo4j is an open source, NoSQL graph management system written using Java and Scala. Neo Technologies sponsors and oversees the development of Neo4j and has a huge influence on its development roadmap. Neo4j is one of the very few ACID compliance NoSQL databases as it uses a proprietary, labeled property graph data model to represent and store data both in memory and at the storage level.

Some use cases for Neo4j include fraud detection, analytics, social networks, recommendations, scientific research, and routing. Since being made open source in 2007, the popularity of Neo4j has increased steadily and has been downloaded more than a million times. Well written guides, video tutorials, and online documentation makes it easy for new developers to adopt Neo4j. The following sections in this chapter will give insights into Neo4j's architecture and will help in the transition from a user to a contributor.

Stakeholder Analysis

"A *stakeholder* is a person, group, or entity with an interest in or concerns about the realization of the software architecture of a system.[1]" This section indentifies the important stakeholders who have a profound impact on the development of Neo4j followed by an analysis of their associated levels of power and interest.

The main stakeholder is Neo Technology, Inc. The organization and its employees exclusively fulfill the roles of four of the eleven classes of stakeholders defined by Rozanski and Woods[1]:- **Assessors, Acquirers, Communicators, and Maintainers.** The most important decisions including architectural design, roadmap, releases are authoritatively taken by the leadership team of Neo Technology, Inc. spearheaded by its CEO, Emil Eifrem. The other stakeholders applicable to Neo4j are listed in Table 1 and Figure 2

Stakeholder	Description
Developers and Testers	Most of the developers and testers are the employees of Neo4j. Employees of major customers including Google, ThoughtWorks, GraphAware are among the top contributors and focus on contributing to specific features to fulfill their requirements. The individual users are more adept at raising concerns and fixing minor bugs.
Suppliers	Though Neo4j supplies the software packages for deployment as downloads, other suppliers provide the whole infrastructure as a service. GrapheneDB specializes in the delivery of managed deployments of Neo4j suitable for most end users. IaaS providers like Heroku, Microsoft Azure supply solutions tailored to meet large scale deployments of Neo4j
OEM Partners	OEM partners specialize in design, optimization and delivery of both the hardware and software together. The solutions provided by the OEM Partners offer the best performance.
Support	Neo Technology's customer support portal provides support for licensed customers. Free support is extended to all via Stackoverflow and Google group
Users	The users are anyone who uses the graph functionality of Neo4j. The broad spectrum of users includes individual software developers, students, educational institutions, government organizations, and enterprises including Linkedin, Walmart, eBay, and Cisco.

Table 1: Other stakeholders of Neo4j.

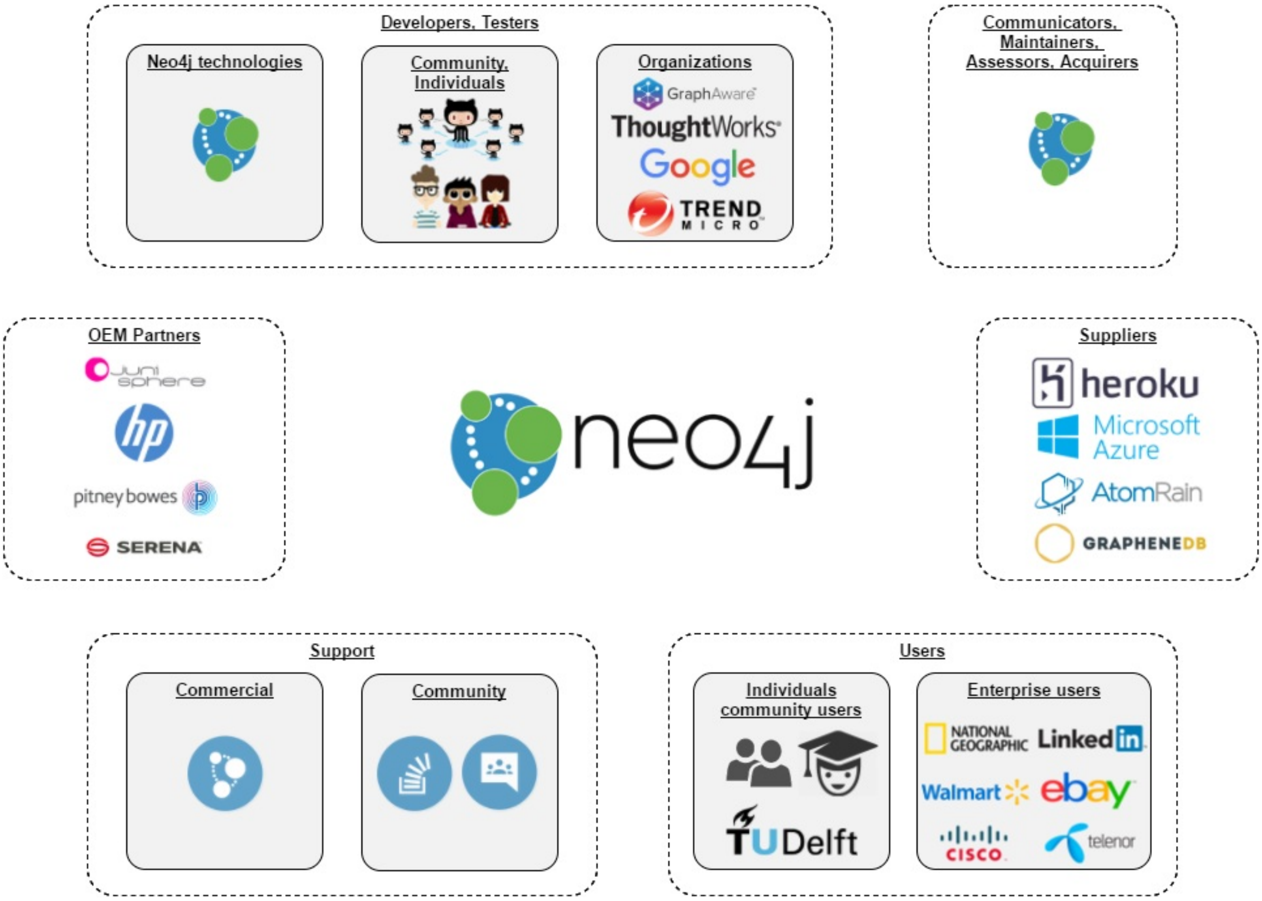


Figure 2: Stakeholders for Neo4j

The power interest graph depicted in Figure 3 gives a visualization of the measure of the power that a stakeholder wields in the system compared to their interest. The bottom-left quadrant has the least power and interest where as the top-right quadrant has the most. The remaining quadrant on top-left and bottom-right bias towards more power and interest respectively.

As the major stakeholder, Neo Technology, Inc. has the most power and interest. Though the OEM Partners and the Suppliers have slightly lesser interest and power, they are still placed in top-right quadrant along with Neo4j. The enterprise customers are vital for the success of Neo4j and have high power with relatively low interest and are placed in the top-left quadrant.

The end users of the system and DevOp specialists have considerably low power and interest and are placed in the bottom-left quadrant. The bottom-right quadrant consists of occasional contributors from the community, regular contributors with relatively more power than the occasional contributors and the competitors. Though the competitors wield no power, their interest is very high as they do regular market and competitor analysis to keep in line with Neo4j's new features.

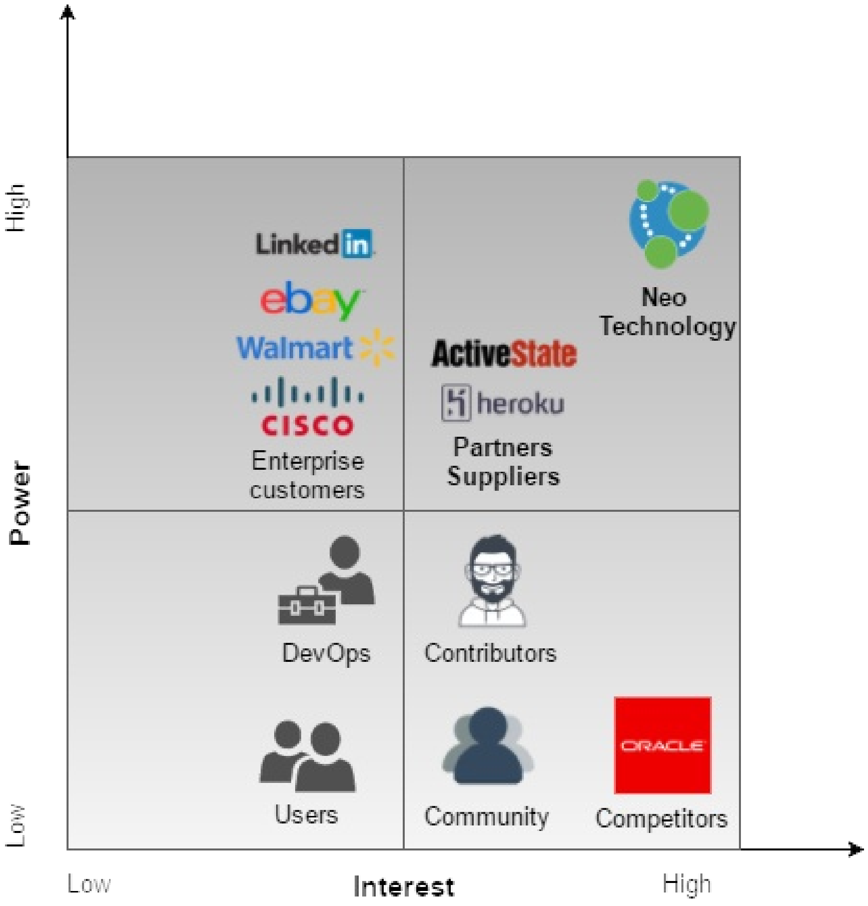


Figure 3: Power/Interest grid for Neo4j

Context View

The context view describes the relationships, dependencies, and interactions between the system and its environment. To this end, this section will determine the scope of Neo4j, analyze the external entities and services that interact with it and finally visualize the relationships uncovered.

System Scope

Databases are used by virtually every company to securely store information in a scalable manner. As databases are set up entirely by the users according to their own needs a database must be able to fulfil many different functions.
In order to be competitive to widely-used existing databases, such as MySQL, Neo4j must be able to meet their standards. However, Neo4j must also provide something that other databases do not. Neo4j does this by utilizing its graph database structure in order to model relationships and store inconsistent data types in far more effective manners.

Thus, the scope of Neo4j is to provide unique capabilities and performance in regards to relationship modelling and inconsistent data type handling, as well as providing all functionalities already present in existing databases.

External entities and interfaces

Neo4j is a fast growing database software with a dedicated company, Neo4j Technologies, sponsoring its development. As can only be expected, there are many external libraries and systems in use to assist in the development and a great deal of interest from third parties. Additionally, Neo4j has many challenges it must live up to as it needs to support a multitude of operating systems and greatly varying requirements. Below, we detail these relationships and visualize them in Figure 4.

- Written in Java and Scala using Eclipse and IntelliJ
- Built using Maven and continuous delivery using TeamCity
- GitHub is the platform used for hosting and maintaining the source code as well as the tracking of issues
- Supports distributions for Windows, Linux & OS X
- Technical help for Neo4j use is supplied on StackOverflow, but developers can also be reached on Slack or even Twitter
- Neo4j has support for drivers in various programming languages including Java, JavaScript, Ruby, Python and .NET
- Dependence on libraries such as JUnit, Mockito, Jetty and Guava, AngularJS, Grunt, Bower, D3.js
- Competes with MongoDB, OrientDB, Titan, Oracle and more
- Dual licences- commercial license for enterprises and open sourced with AGPL3
- Used by small, and large organizations including Cisco, Walmart, National Geographic and more.
- Partnered with providers like Heroku, Microsoft Azure, GrapheneDB, ActiveState to provide customized solutions

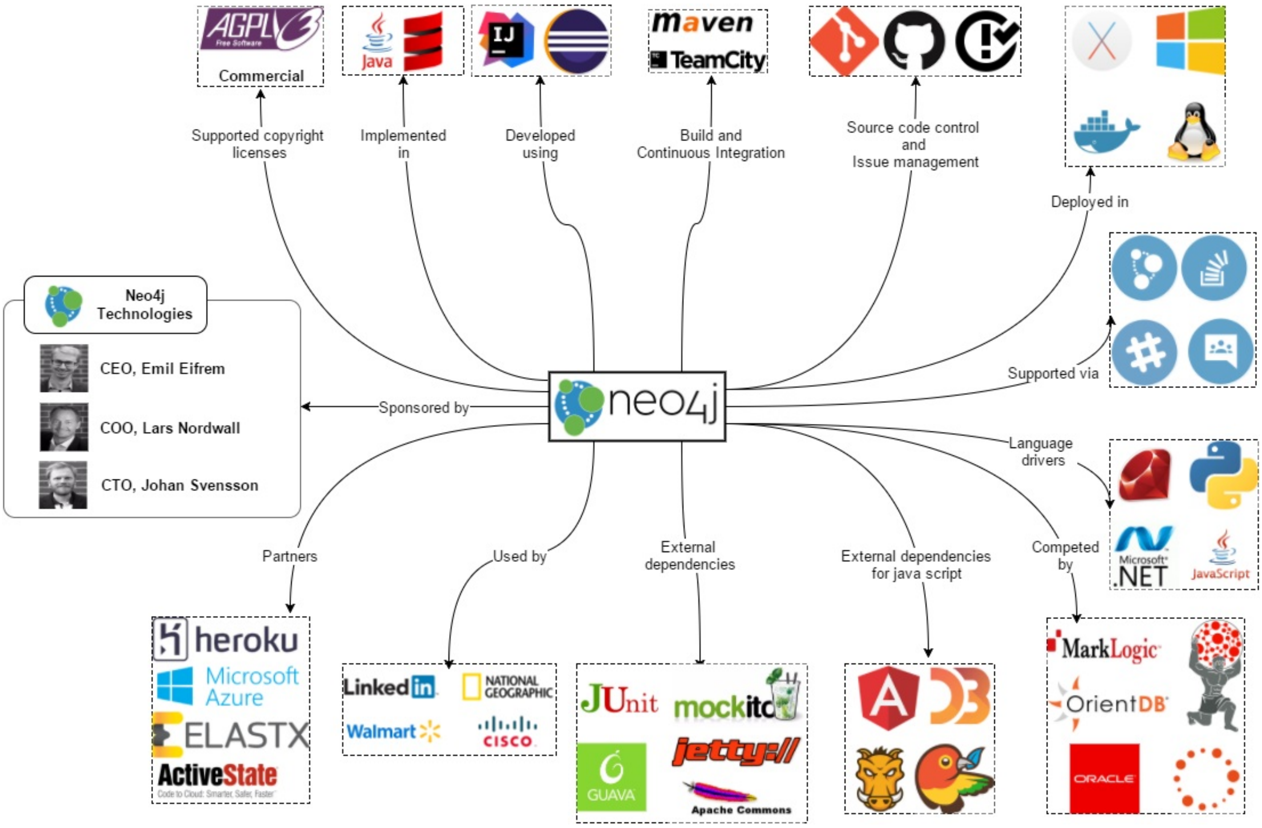


Figure 4: Visualization of Neo4j's Context View

Development View

The development view of a system describes the architecture that supports the software development process. The following section address the concerns of a developer like module organization, common processes, standardization of design and testing, and codeline organization. Finally the technical debt of Neo4j is discussed.

Module organization

The module structure model deals with the organization of system's source code in terms of modules[1]. The components of the two distributions of Neo4j- community, and enterprise are organized in distict folders. Logically, the enterprise edition encapsulates all the modules in community edition in addtion to a few extra components. This section focusses on structure and organization of the code for community edition of Neo4j in three abstract layers as shown in Figure 5.

The different components of Neo4j community edition can be organized three logical layers as seen in Figure 5 in distinct layers

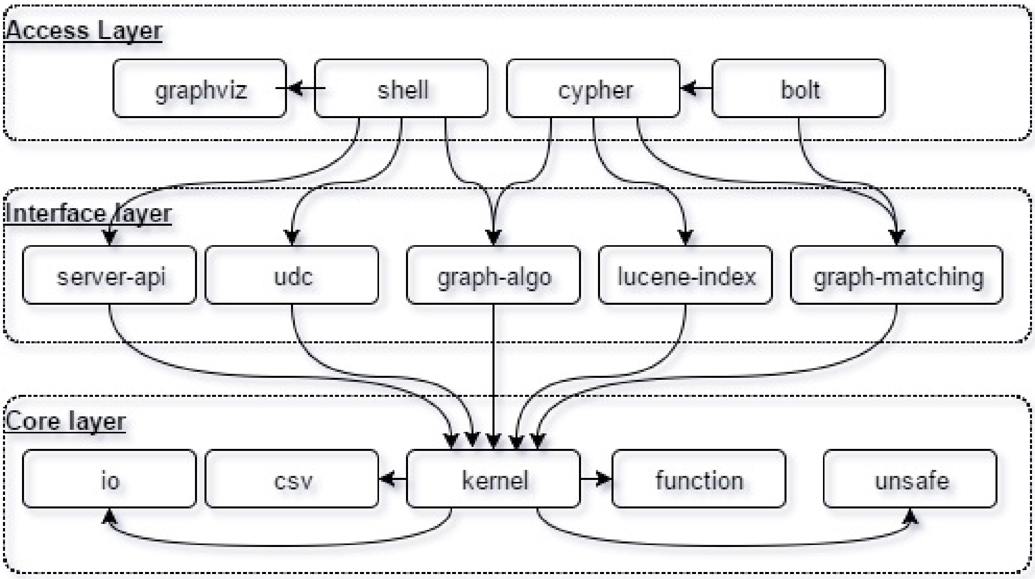


Figure 5: Module organization of community edition

1. **Access layer** - Provides external interfaces to the database system including cypher query language, bolt protocol, shell and graphical visualization.
2. **Intermediate layer** - core functionalities of the graph system including server APIs, graph engine, graphing algorithms, data collector, lucene index.
3. **Core layer** - made of internal components that are not accessable outside the system. Kernel forms the core of the Neo4j platform priving access to storage and memeory. This layer consists of components like io, csv, function primitives, and unsafe memory access.

Common Processing

Neo4j is highly modulazired with minimal code duplication. Some common modules that reduce the code duplication include `csv` , `io` , `collections` , and finally `common` . The `common` foler contains most of the reused code and greatly helps in code maintainance.

Standardization of design

Being an opensource project, Neo4j has a set of strict guidelines for making contributions.

- The contributor must sign CLA.
- Use Eclipse or IntelliJ for development
- Standardized templates are available for raising issues in GitHub.
- The code should strictly adhere to style checks defined and available for Eclispe or IntelliJ IDEs.
- The commits must be squashed down and rebased without any merges.

- Documentation should be updated for any code changes.

Standardization of testing

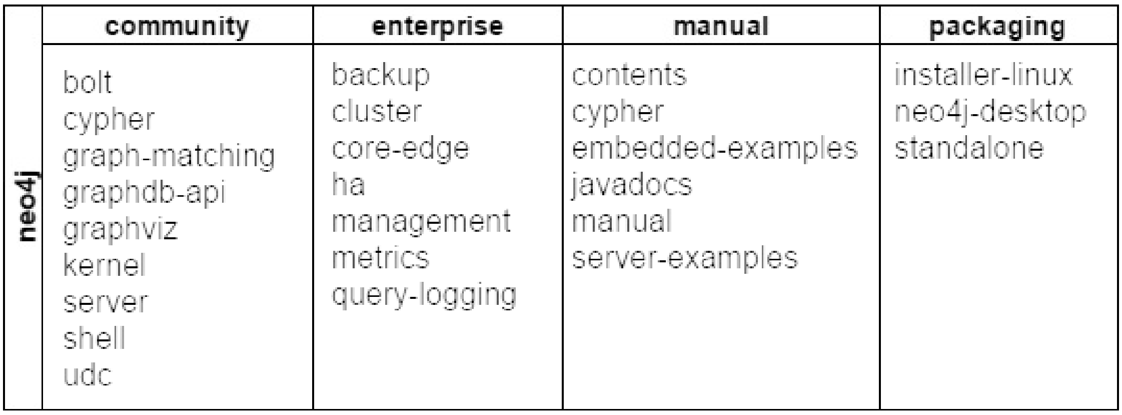
The maven build process automatically triggers and executes all tests defined in Neo4j's source code. Code contributions are allowed to be merged only if the tests succeed. Tests are written using Junit and mockito. Continuous integration is achieved by using TeamCity. Following guidelines must be adhered for a Pull request to be successfully merged:

- All tests must succeed.
- Code coverage computed by cobertura is within acceptable limit.
- TeamCity integration tests succeed and no issues are raised by CI process.

Codeline Organization

Codeline model defines the overall structure of the codeline and ensures that order is maintained in the organization of the system's code[1]. The source code of both the editions are maintained in the same github repository. In Figure 6, the top level folders are displayed in swim lanes along with their components in respective lanes.

The important folders at the root of the Neo4j repository are `community` , `enterprise` , `manual` , `packaging` , and `tools` . `manual` contains documentation of Neo4j in AsciiDoc format. It can also extract documentation from source code. `packaging` contains the various utilities for packaging and distribution of the Neo4j releases for different operating systems. Each component inside the community or enterprise folder has a maven `pom.xml` file in addition to `src` folder consisting of Java or Scala source code in `main` , along with `tests` , and `docs` .



	community	enterprise	manual	packaging
neo4j	bolt cypher graph-matching graphdb-api graphviz kernel server shell udc	backup cluster core-edge ha management metrics query-logging	contents cypher embedded-examples javadocs manual server-examples	installer-linux neo4j-desktop standalone

Figure 6: Swimlane view of important directories of Neo4j

The main components of the Neo4j distribution are listed in table 2.

Directory	Description
cypher	Cypher execution engine to support cypher query language
graph-matching	Graph pattern matching APIs
graphviz	Visualization of graph data.
kernel	Core of Neo4j that contains the storage system, embeded API, traversal API, batch processing, configuration, locking and other utilities.
shell	Provides a command line interface to Neo4j
backup	Create backup of Neo4j database over the network
ha	Enables fault-tolerant database architecture and supports replicas as slaves.
metrics	Modules to expose and report Neo4j metrics

```
Table 2. Important components of the Neo4j
```

Technical Debt

Technical debt describes the occurrence of using quick and cheap methods of fixing bugs or implementing features, methods which are typically not thought through well. In essence, the problem with performing such actions lies in the manner in which they make future work more difficult. This is because this manner of work typically does not follow the existing architecture or system design, making it more difficult to oversee the functionality of the system as a whole and create future changes.

Analysis of Neo4j's technical debt yields very good metrics; only 0.5% of classes have flaws in them. Interestingly, the two classes with the highest cumulative class flaws are both test classes, namely `PageCacheTest` and `NeoStoresTest`. The largest package in Neo4j kernel, `impl` package has likely gotten too convoluted over time and many problems occur within it.

Technical debt is typically differentiated using 'Reckless' vs 'Prudent' and 'Deliberate' vs 'Inadvertent', forming a total of 4 quadrants together. Neo4j contains relatively few flaws, leading us to categorize Neo4j as 'Prudent'. Additionally, the area that is most flawed appears to be the a part of the core functionality. This is likely a result of many years of development despite best attempts to avoid it, leading us to also categorize Neo4j as 'Inadvertent'.

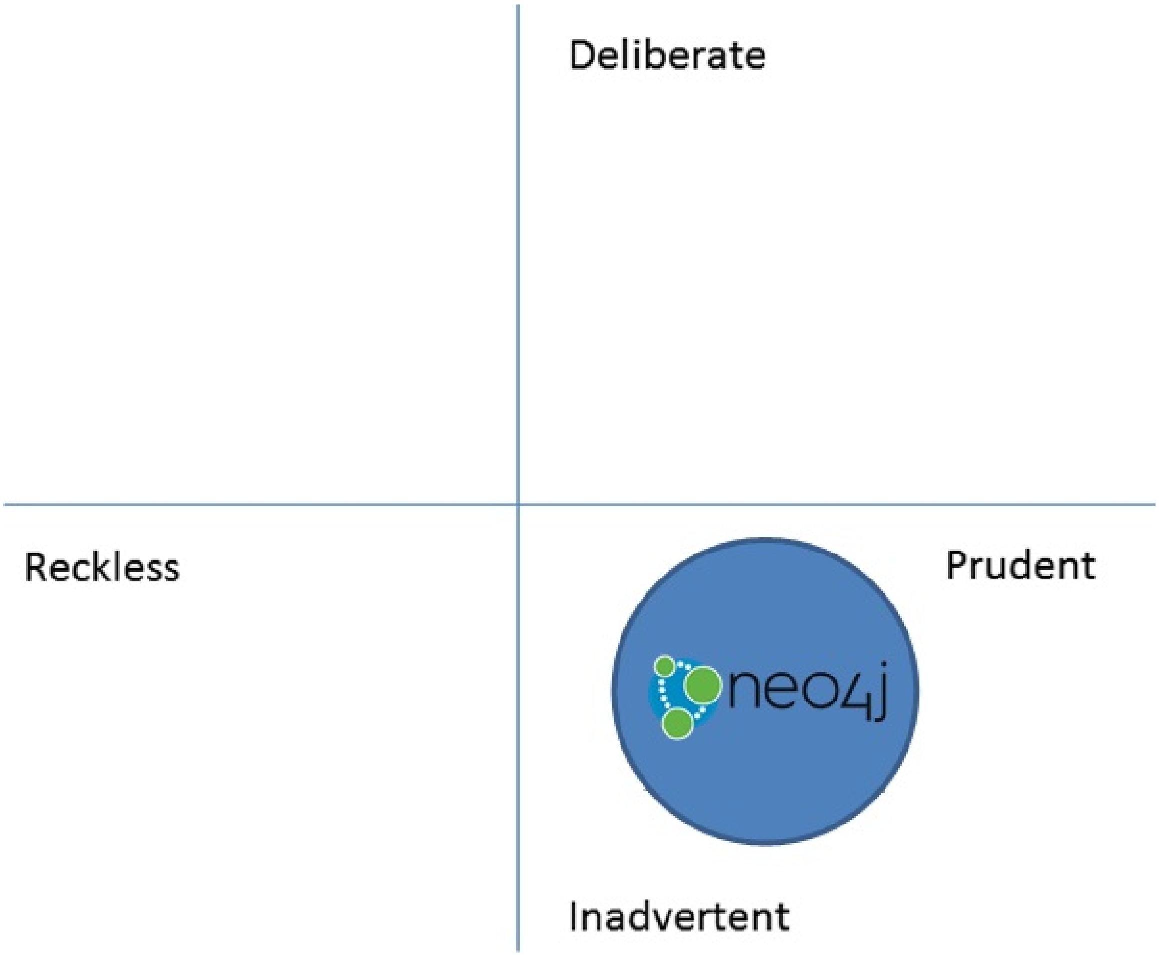


Figure 7: Technical debt quadrant

Deployment View

Database systems are a thing which every modern company requires. This makes it all the more important that database software is simple to deploy and does not impose too many constraints on its users. Neo4j works towards this from the very beginning by utilizing Java and Scala, which can run on any Operating System, allowing users to choose the Operating System which they are most comfortable working with. Additionally, the hardware requirements are very lenient, being able to work with as little as 2 GB of RAM and 10 GB of disk space. In this section, we will look further, however, and analyze third-party software requirements as well as network requirements which Neo4j needs to function.

Third-Party Software Constraints

The installation of Neo4j requires no third party software other than Java runtime 7 or 8 (preferred). However, there are several third-party programs which Neo4j natively supports and that a user may want to install. Below, these programs are highlighted.

- It is recommended to install software that can visualize the graph database. Neo4j recommends the use of commercial product Linkurious. However, open-source software Gephi is frequently used for this as well.
- Though it is possible to interface to a Neo4j server using the Neo4j Shell, the recommended method is to use an internet browser such as Google Chrome, which allows it to use the D3.js library to visualize data to the user.
- Neo4j supports the use of Docker, a program that can wrap another piece of software in a complete filesystem. Using Docker it becomes simpler to get Neo4j servers running on multiple systems, as it can store the image of the software and install it on another computer with no further hassle. The software on both systems will be identical.

Network Requirements

Neo4j itself uses native messaging and does not have any specific requirements other than interfacing with it using Neo4j Shell or an internet browser. However, the High Availability module, which is responsible for database replication in a master-slave configuration, does impose certain network capacity requirements. As this module is responsible for ensuring identical data across several Neo4j servers, this synchronization can put strain on a network. However, Neo4j has numerous settings to manage the functionality of the High Availability module. For example, the `ha.pull_interval` setting determines how frequently slaves pull updates from the master. Because this module can be fully configured as per the needs of the user this module has no exact network requirements, but should be taken into consideration when designing and running your database.

Evolution Perspective

In order to study the evolution of Neo4j's configuration and variability[1], we analyzed changelogs and release notes available at Neo4j's GitHub wiki. Neo4j's evolution history in regards to configuration and variability has been fairly minimal, with the only major changes actually being in recent history. The following section details the changes to configurations in the most recent releases of `2.2` and `2.3` , and `3.0` . For the purpose of keeping the lists clean and concise, bug fixes related to settings are not included.

Evolution in version 2.2

- Query log file rotation was added using the `dbms.querylog.rotation.threshold` option (this is the option mentioned earlier in the feature identification).
- `dbms.querylog.filename` was changed to specify a filename rather than a directory, as would be expected from the name. Additionally, the default value was changed to

`null` , meaning the query logger is disabled by default.
- `logical_log_rotation_threshold` default value was changed from 1MB to 250MB.
- `mapped_memory_total_size` was renamed to `dbms.pagecache.memory` .

Evolution in version 2.3

- The default setting of `dbms.pagecache.memory` was changed from 75% to 50% of free system memory
- As a result of the Object cache being removed the following settings were removed: `cache_type` , `node_cache_size` , `relationship_cache_size` , `node_cache_array_fraction` , `relationship_cache_array_fraction` , `cache.memory_ratio` , `high_performance_cache_min_log_interval`

Evolution in version 3.0

In the latest release, version `3.0.0-M04` , a change was made to the way in which the configuration was loaded. The old functionality had two configuration files, `neo4j.properties` , and `neo4j-server.properties` . However, as the difference between these was sometimes vague, as well as the two separate files being rather clumsy, it was merged into the singular `neo4j.conf` that was mentioned last section.
In the process of this update, a number of system properties which Neo4j used to find the configuration file(s) were changed, as well as the developers taking the liberty to remove various deprecated configuration settings from older versions (primarily version `2.2`).
Furthermore, the following changes occurred:

- The cypher pre-parsing query planner option `greedy` was removed
- The cypher pre-parsing update strategy option `eager` was added.
- The setting `org.neo4j.server.database.location` was removed.
- The setting `dbms.active_database` was added, more or less a relocation of the former setting.
- The setting `dbms.directories.data` was added, to specify where the database stores its data.

To conclude, while there are certainly changes in the configuration options of each Neo4j version, considering the release timeline there is a fairly minimal amount. This can most likely be attributed to a large part that Neo4j's development focuses on improving performance and adding new features (which do not require many, if any, new settings).
There is a fairly small amount of deprecated settings as most modules that have been added over the duration of Neo4j's development time have not been removed or replaced, which can be attributed to Neo4j having a clear and detailed design architecture which the developers follow closely.

Variability Perspective

The variability of a system is an important characteristic of any system. Through customization of the settings a system can be made easier to work with for users, performance of the system can be increased and unused or unwanted modules can be disabled. In short, by providing a great deal of customization options a product can gain a competitive advantage over similar products.

Neo4j, being a database system, has a particularly high need for variability. It serves as a framework for users of Neo4j to store all data they deem important and can have many different use cases. For example, a bank requires their database to have features and options to tighten security and confidentiality as much as possible, where as a cloud-service requires fast responses and high availability.

These demands can be met using variability, which is especially ideal as making all such features compulsory would mean overburdening users that have no need for such advanced features. In this chapter we identify and analyze the work that Neo Technologies has put toward variability based on the work done by Apel et. al in their book *Feature-Oriented Software Product Lines*[2]. We analyze the ways in which Neo4j can be customized by its users, as well as the way in which these customizations were implemented by the developers.

Feature Identification

In this section we identify a number of the configuration options (features) available in Neo4j. In total there are roughly a hundred options, ranging from basic settings such as specifying the location of the database directory, to more advanced optimization options such as buffer and cache sizes. Settings are either general settings or module-specific settings. This section gives a number of examples of both general settings and for a specific module, in this case, Cypher was chosen.

General settings

A number of the settings available in Neo4j do not belong under any specific module, but are still significant settings which are important to mention. Below a number of these are highlighted.

- **Allow File URLs** - A server setting that determines whether the Cypher API will accept file URLs when loading data using `LOAD CSV`.
- **Read Only** - This setting determines whether users are allowed to write new data or update existing data to the Neo4j server or not. This is a useful method for developers to protect their data from users if the users have direct access to the server.

- **Index Sampling Update Percentage** - Determines the percentage of indexes need to be updated before sampling of an index is triggered.
- **Internal Store Log Level** - Sets the verbosity level of the log. Can be set to e.g. `DEBUG` , `INFO` , `NONE` and more.

Cypher settings

Cypher Query Language (CQL) is Neo4j's custom made query language, similar in nature to the more well known SQL. There are a number of settings specifying how cypher queries are processed and handled, expanded upon below.

- **Cypher Parser Version** - Sets the language version to use of the Cypher parser. Must be either `2.3` or `default` .
- **Cypher Planner** - Sets the default query planner to use which determines how to execute and optimize the query given to Neo4j. Should be either `COST` or `RULE` .
- **Cypher Min Replan Interval** - Determines the minimum time before Neo4j will start considering to re-plan a query currently in execution. This is typically done because the initial plan was poorly constructed and would take longer than re-planning a more efficient plan.
- **Query Cache Size** - Determines the maximum number of Cypher query execution plans that the server keeps in cache.

Feature Relationships

The majority of features in Neo4j possess no dependency and have no conflicts with each other besides the need to turn their respective module on. For example, for the 'Query log threshold' setting to have any impact, 'Query logging' itself must be turned on.

Furthermore, though many settings are mandatory (i.e. the system always uses them), every setting has a default value. Most of the time this default value need not be changed. In fact, in case of the 'High Availability' module it is recommended that you do not change anything. As such, despite there being a great deal of settings the amount being requested of the user is very minimal.

Figure 8 displays the relationship between a large number of features present in Neo4j. Unfortunately, there are too many to display all within a single figure.

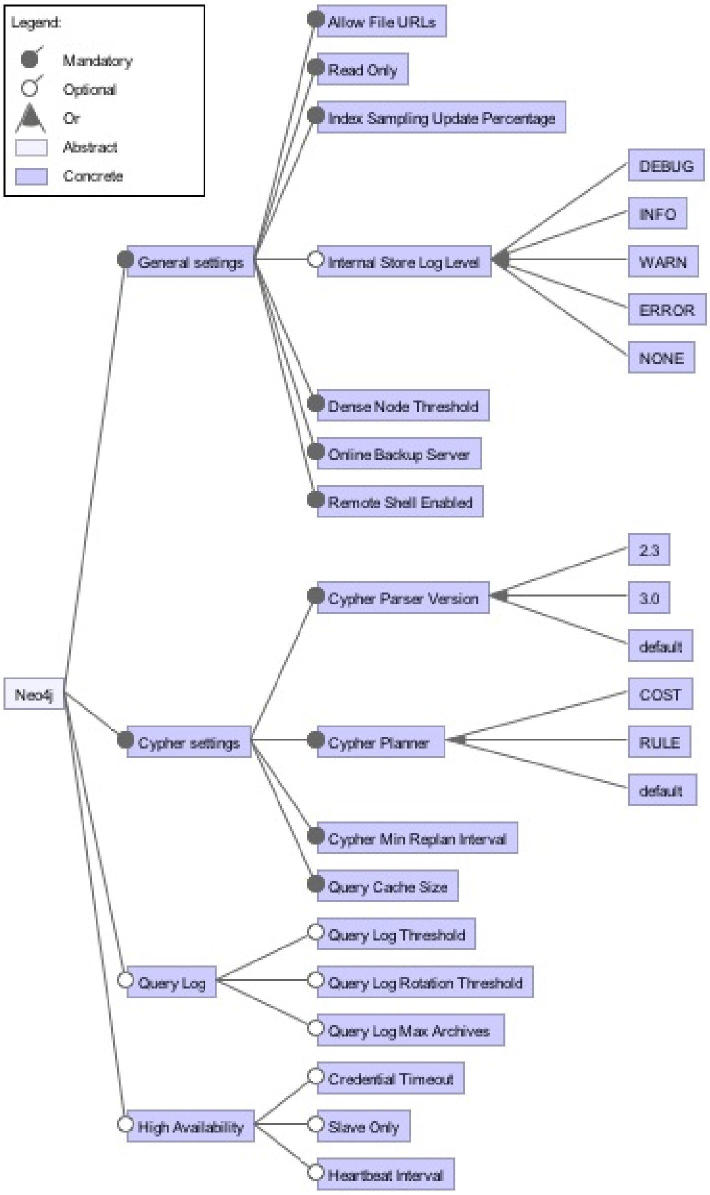


Figure 8: Feature relationships in Neo4j based on [3]

Feature Binding Time and Variability Strategy

All Neo4j configuration options have a binding time of startup-time. The reason behind this decision is most likely a result of the manner in which databases are typically used, namely as long-running programs, often being active for days, weeks or even months at a time without pause. During this time there is generally no need to change settings, hence the ability to do so was not considered important.

The strategy for implementing these settings was to use a single configuration file. All settings can be defined in a `neo4j.conf` file which is located within the base directory of a database. This configuration file is read upon start-up of the Neo4j server and the settings are then stored within the program itself as a global object for quick access by the program. As such, making modifications to this file after start-up will not affect the server until Neo4j has been restarted.

The configuration file has a simple structure, in which settings are assigned values with simple `=` statements and comments can be placed using `#`. For example, below is what a possible configuration file might look like:

```
# Enable shell server so that remote clients can connect via Neo4j shell.
remote_shell_enabled=true
# The network interface IP the shell will listen on (use 0.0.0.0 for all interfaces).
remote_shell_host=127.0.0.1
# The port the shell will listen on, default is 1337.
remote_shell_port=1337
```

By default the properties file is empty, meaning that each setting uses its default value. By giving each setting a default value Neo4j becomes easier to use, as less specific information is requested from newly starting users. The default values that are used for each setting can be found in the Neo4j documentation, which also features a description of each setting and the possible input values (e.g. value ranges/enumerator values).

Performance and Scalability Perspective

The performance and scalability is very crucial to a database. One of the most important characteristic to evaluate the performance of a database is the query time. Due to the tremendous rate of growth of data in the present day, it becomes more difficult for traditional database to provide real-time feedback. Furthermore, the increase in data means that databases should have be highly scalable to fit the needs. The graph database that was created and implemented by Neo4j has achieved that goal in its current situation. However, will Neo4j still meet the performance criteria if the workload increases even further in the future? The following sections focus on further discussion of the performance and scalability of Neo4j. Since these two points are essential to this software, we want to spend some time to analyze it on a number of points.

Performance

There are two major characteristics of performance:

1. Response time
2. Throughput

The Neo4j is known as high performance when managing a huge amount of different data. Relational database search all of the data looking for anything that meets the search criteria. The larger the set of data, the longer it takes to find matches, because the database has to examine everything in the collection. However, a graph database only examines at nodes that are directly connected to other nodes. If a limit is given on how many steps it is allow to make, it can ignore everything further away for additional performance boosts.

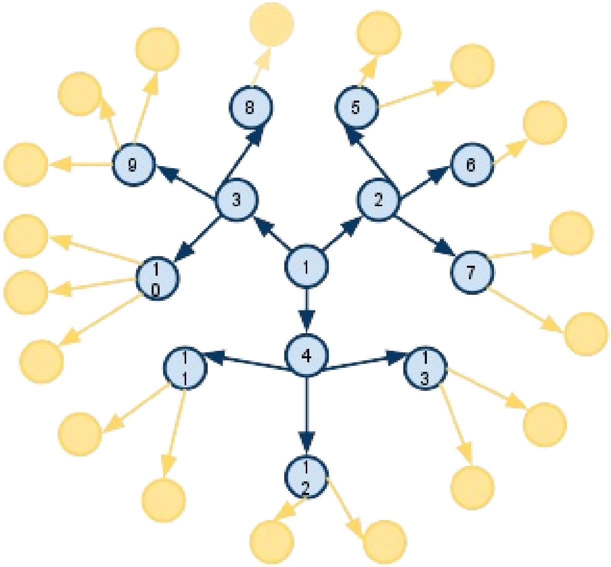


Figure 9: Visualization of Neo4j's node routing

Figure 9 depicts how nodes are connected within Neo4j. Only the blue nodes are searched during a query. Neo4j knows where it is at any time, and has no need to start over from the beginning or backtrack.

Response time

This leads to the difference in response time compared to relational databases. Graph databases have a much better performance if the data is not identical. The average response time will be hundreds times faster than the relational database. Conversely, it will be slower if the data is identical.

Throughput

The throughput of Neo4j will vary considerable depending on the data type. If the data is unique and the steps limit is set, the performance will be quite similar. However, the performance will be heavily influenced if the numbers of connections between two nodes increases drastically.

Scalability

Scalability can mean different things to different people. Common points are:

1. Redundancy to failure
2. Managing increasing read load
3. Managing increasing data size
4. Managing increasing write load

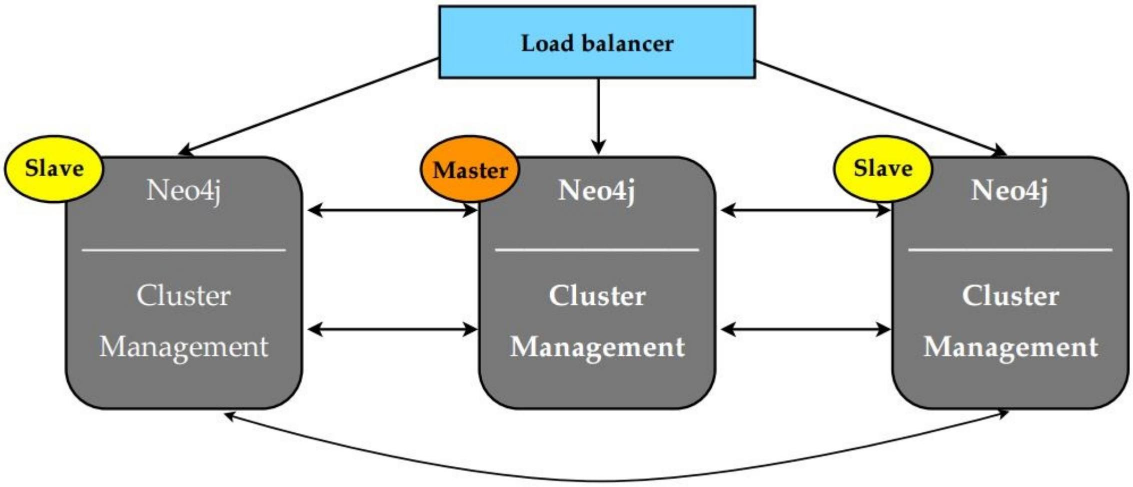


Figure 10: Visualization of Neo4j's Cluster system

Redundancy to failure

Neo4j offers a scalability package which includes:

- Online backups when the cluster is running
- Global cluster for data locality
- Disaster recovery for data center redundancy
- Reporting instances for ad-hoc reporting

In a Neo4j cluster, the full graph is replicated to each instance in the cluster. All the data is safe as long as one instance remains available. A single instance of Neo4j can house at most 34 billion nodes, 34 billion relationships, and 68 billion properties, in total.

Managing increasing read load

Read operations can be done locally on each slave. This allows the read capacity to increase linearly with the number of servers.

Managing increasing data size

A graph database can find the neighbors of any node without going through the all relationships. Thus, as the data size increases, the query time will remain a constant. Additionally, if the steps limit is set properly, the query time is likely within a very small range. However, the performance will be influenced depending on whether the data is unique or not.

Managing increasing write load

Neo4j has a single master to coordinate all write operations, and it limits the write throughput of a single machine. However, there are few scenarios dealing with high write load. Furthermore, a queuing solution could be implemented to handle this situation. A steady manageable stream of write operations can be serviced by the cluster.

Conclusion

This chapter summarized Neo4j in architectural opinions. The stakeholder analysis was discussed followed by different views and perspectives. In the stakeholder analysis, who they are and what they do were discussed. After that, there were three views introduced the software in context, development and deployment. At the last, there were three perspectives introduced the software in evolution, variability, performance and scalability.

Companies and organizations after reading this chapter can get more insght to the software. Companies and organizations today not only need to store the large amount data, but need to get an insight from the existing data as well. In order to use the data relationships, companies and organizations need a database to store relationships as well.

Neo4j is a new approach to cope with the increasing data. Also, graphs are the most efficient and natural way to represent relationships. It gives higher efficiency, faster response, easier maintenance and safer operations comparing to the conventional database.

Bibliography

[1] - Rozanski, Nick, and Eóin Woods. Software systems architecture: working with stakeholders using viewpoints and perspectives. Addison-Wesley, 2012.

[2] - Apel, Sven, et al. "Feature-Oriented Software Product Lines."

[3] - Thüm, Thomas, et al. "FeatureIDE: An extensible framework for feature-oriented software development." Science of Computer Programming 79 (2014): 70-85.

[4] - Neo Technologies' CEO, Emil Eifrem's interview
https://twitter.com/emileifrem/status/712327903032188928

OpenCV (Open Source Computer Vision Library)

Bjarki Johannsson, Shruthi Kashyap and Renukaprasad Manjappa

Delft University of Technology

Abstract

OpenCV is an open source, cross-platform, cross-language computer vision library, that was devised in order to create a common infrastructure for computer vision applications and to expand the usage of computer vision in commercial products. The library consists of more than 2500 optimised algorithms and supports hardware accelerators such as CUDA and SSE. This chapter provides a concise overview of the OpenCV project by presenting an analysis of the project through multiple perspectives and viewpoints as presented in the book by Rozanski and Woods [4].

Table of Contents

Introduction

Computer vision is a discipline, that deals with acquiring, processing as well as analyzing images. Based on these functions, computer vision tries to extract as much useful information out of the images as possible, which can be used to make decisions. A widespread theme has been the search for methods that could provide computers with human-like abilities in understanding images and deducting useful information out of it. Computer vision is frequently used in autonomous driving, object recognition as well as product quality management amongst others. [15]

Computer vision has gone through a significant growth during the past decades and nowadays there are many libraries available for computer vision applications. The widespread usage of computer vision has been made possible by the combination of more capable algorithms, cheaper and more powerful hardware and better cameras. [3]

OpenCV (Open Source Computer Vision Library) is a widely used computer vision and machine learning library mainly aimed at real-time applications. It was originally developed by Intel employees in a research center in Russia, but the project was taken over by a non-profit foundation in 2012. [1, 2, 3]

The library is written in C and C++, but it is intended to be a cross-platform, cross-language library, therefore supporting languages such as Python, Java and operating systems such as Windows, Linux, Mac OS X, Android and iOS amongst others. The algorithms have been optimised in C and take advantage of multi-core processors and GPUs. [1, 2, 3]

OpenCV has been licensed so that enterprises can use it in their products without providing the source code of their applications. Therefore, OpenCV has been also used by many corporations, such as Toyota, Google and IBM. [1, 3]

The main motivation for starting the OpenCV project back in 1999 were the following points [3]:

- Advance the computer vision field by providing optimised code.
- Spread knowledge of computer vision by providing a common library, that would make it easier to develop new applications.
- Make commercial applications more viable by making the optimised code free, without any limiting licenses.

The chapter has been divided into 6 sections. The chapter starts with an analysis of the stakeholders, which is followed by the context view. These sections provide the main background information needed to understand OpenCV. The development view and evolution perspective provide an overview of the source code structure as well as how it has evolved during the long history of OpenCV. Finally, the deployment view as well as the performance and scalability perspective describe how OpenCV is used at run-time, what features it uses in order to increase performance and how scalable it is.

Stakeholders

Stakeholders are the individuals, teams or organisations that have an interest in the realization of a software system [4]. We have found 5 types of stakeholders applicable to OpenCV out of the 11 types presented in the book by Rozanski and Woods [4]. Most of the stakeholders are not applicable to OpenCV, since the book considers the design of a corporate web application, but OpenCV is an open source platform-specific library. The OpenCV stakeholders are presented in Table 1.

Stakeholder	Description
Communicator	Maksim Shabunin has provided systematic work on documentation, but otherwise it is maintained by the whole community.
Developer	Alexander Alekhin, Maksim Shabunin, Steven Puttemans are the main developers, but more than 533 users in total have contributed to the project. Most of the developers have contributed, because they have encountered an issue while using the library and have come up with a solution to fix it. They take action in order to influence the next releases as well as to help other developers who might encounter the same issues.
Tester	Testers are not explicitly used, instead the developers write the tests when implementing new functionality and an automated bot, OpenCV Pushbot, is used to run the automated tests when an integrator has approved a pull request.
User	The users are comprised of small and large corporations, government agencies and software engineers. Some examples of corporations using OpenCV are Google, Intel, IBM and Toyota, but the total number of downloads has been estimated to more than 7 million [2].
Integrator	Alexander Alekhin, Maksim Shabunin and Steven Puttemans are the integrators. They try to develop the project in a systematic manner while maintaining the code quality, code style and the architecture. It seems that they are clearly struggling with invalid and duplicate issues, which make up roughly half of the recently closed issues.

Table 1: Stakeholders within the OpenCV project.

Context View

The context view defines the relationships, interactions and dependencies between OpenCV and its environment. This view gives a broad overview of the whole system and is therefore useful for all stakeholders. The context view model for OpenCV is presented in Figure 1.

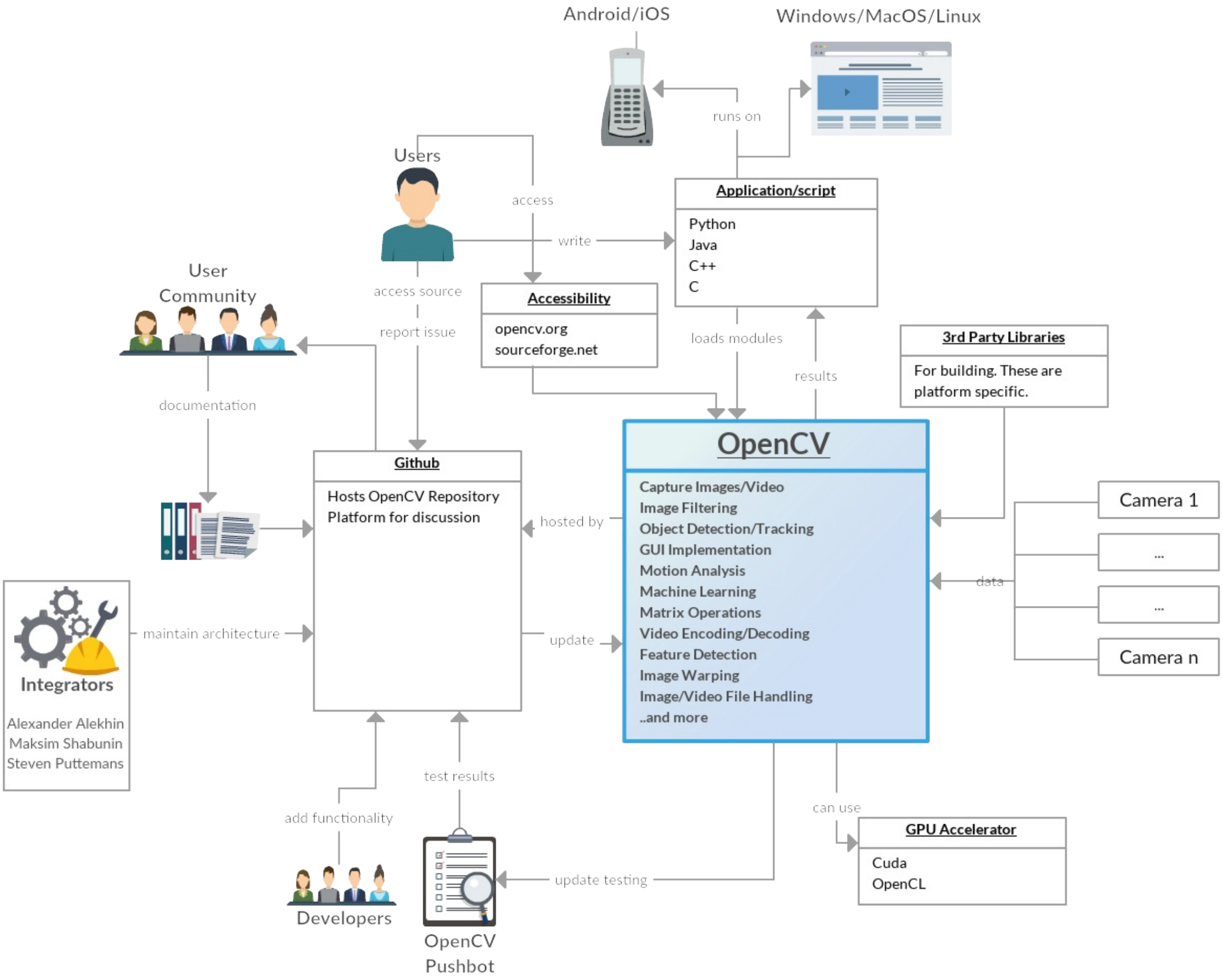


Figure 1: Context view model for OpenCV.

OpenCV is tightly coupled with GitHub, which hosts the source code and where users can clone, fork or download it from. Issues and discussions about missing or faulty functionality also take place on GitHub. Developers can contribute to the project by making pull requests, which are then handled by the integrators. Documentation is provided by the user community through GitHub, although Maksim Shabunin is the main contributor in this category. A binary distribution is provided on the OpenCV web page and on Sourceforge for developers who have no intentions on modifying the source code. All of the stakeholders discussed in the Stakeholders section are also covered in the context view model and all of them make use of GitHub as a means of accessing information about OpenCV.

OpenCV is a real-time, highly optimised library, therefore it needs to have support for many hardware accelerators and external devices. It has support for CUDA, a platform for parallel computing on NVIDIA's GPUs. Over 375 million CUDA-enabled GPUs are present in modern computers, which allow the developers to run their algorithms on GPUs. OpenCV also supports another parallel programming library, namely OpenCL (Open Computing Language), which is an open source, royalty-free standard for parallel programming from the Khronos Group.

OpenCV is a cross-platform, multi-language library with support for all of the major operating systems as well as the C++, Java and Python programming languages, therefore it has to make extensive use of many third party libraries. It has support for many cameras, which are the main data sources for computer vision application via third party camera interfaces and libraries.

Development View

The development view describes the architecture of OpenCV as well as the surrounding build, release and configuration management procedures. The development view gives a broad overview of the source code and is therefore mainly oriented towards software developers.

Module Structure Model

The source code for the OpenCV library is located in the *modules* folder in the repository structure (refer to Source Code Structure for details on the structure). The OpenCV library is huge, containing support for more than 2500 algorithms. The library has evolved for more than 15 years and has seen a tremendous growth in size during this period.

There is no apparent structure amongst the modules, which could be detected from analysing the structure of the files. Nevertheless, there is an enormous amount of dependencies between the modules, which can be detected from the *CMakeLists.txt* files in each module. These modules together with their dependencies are presented in Figure 2.

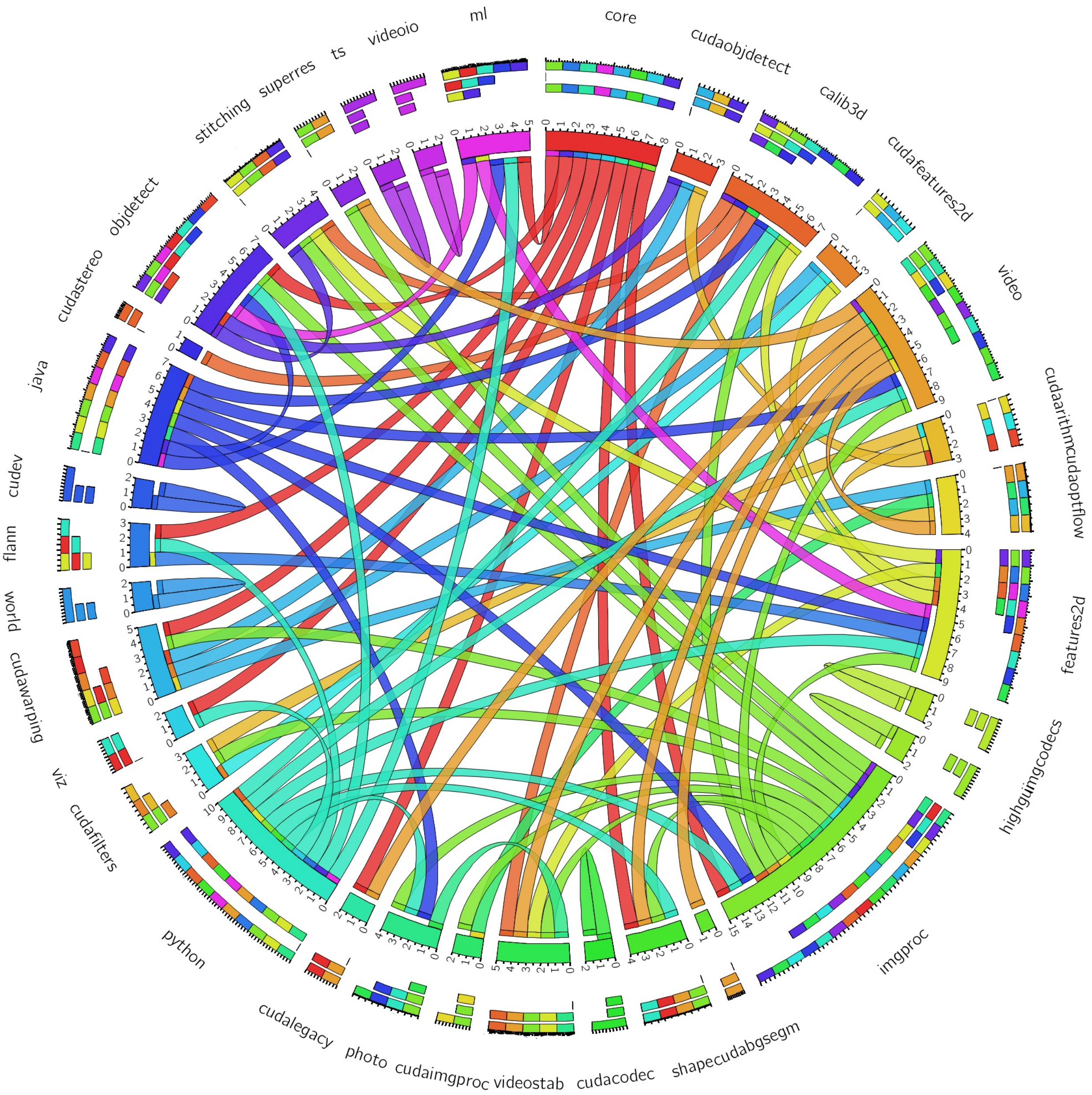


Figure 2: Inter-module dependencies of OpenCV.

Module Hierarchy

Although there is no apparent structure between the modules, it can be easily observed that the common modules that other modules depend on are *core*, *python* and *java*.

The *core* module contains the main algorithms and implementations, which are used the most in the library. This module contains most of the common design models, which reduce duplication.

The OpenCV algorithms are implemented in C++, but those algorithms can also be called from Python and Java. The *python* and *java* modules are used to automatically create wrapper functions for Python and Java from the C++ implementation. Therefore, these modules are commonly referenced from most modules [5].

Apart from the *core*, *python* and *java* modules, there is only one additional module, which sees widespread usage amongst other modules, that is *imgproc*. Other modules create a complex network of a few dependencies per module, that are hard to categorise into largely used common models.

In general, there is no layering in the hierarchy of the modules. While *core*, *python* and *java* are the common modules, there isn't a clear separation above them. It is common that a module that refers to the *core* module also refers to another module that depends on the *core* module, therefore creating a very complex structure.

Software Languages

The total number of source code lines (excluding blank lines and comments) in the *modules* folder was 451,756 at the time of writing, and 1,452,059 in the whole project. The most widely used language in the OpenCV modules themselves is C++, as can be seen from Figure 3, which depicts the languages used in the OpenCV modules. Nevertheless, there is a significant amount of CUDA and OpenCL as well as Java and Python code, which is needed to improve the performance or portability. Furthermore, many other languages are used only for very specific tasks.

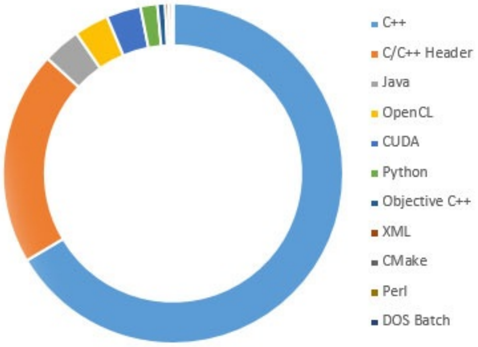


Figure 3: Software languages used for the development of the OpenCV modules.

Surprisingly, the most widely used language when considering the whole project is XML, which has 667,276 lines of code. Furthermore, the total number of languages used in the project is 28, which includes a significant amount of XML, YAML and CMake amongst others. The large amount of XML and CMake code clearly shows the significance of automation in the OpenCV project. We believe that this variation in the languages used makes the OpenCV project very intriguing, but also very complex for newcomers to master.

Guide for Developers

The OpenCV coding style guide [6] describes the main ideas behind the file and folder structure. The source code for the library is located in *opencv/modules/moduleName/src*, the headers are in *opencv/modules/moduleName/include/opencv2/moduleName* and the tests are located in *opencv/modules/moduleName/test*. The guide suggests creating a new module in the modules folder, whenever it seems appropriate. This has led to a large number of modules in the library, that are placed in parallel in a single folder.

Codeline Model

The codeline model deals with the organisation of the source code in folders as well as the build and release procedures.

Source Code Structure

The OpenCV library has been organised into 9 folders, which are depicted in Table 2 [7].

Folder	Description
3rdparty	Contains third party libraries used by OpenCV
apps	Applications used for development
cmake	Contains the core CMake files, that are used for configuration management
data	Contains files that are used by the library, demo applications and testers
doc	Contains the documentation, that is also available on the OpenCV web-page
include	List of included modules
modules	Contains the OpenCV library
platforms	Contains the tools and files needed for cross-compilation
samples	Contains the sample code for library usage

Table 2: Source Code Structure.

Build Approach

The OpenCV project uses CMake to manage the build process. Therefore, the library is highly configurable, allowing the developers to customize the library by excluding unused modules, building the debug or release version, building with or without the support for CUDA and OpenCL etc.

The binary can be compiled after configuring the system with CMake by simply executing Make. This makes the build process straightforward, therefore limiting the amount of build steps although this is a multi-platform library [8]. OpenCV can also be cross-compiled to other platforms. This has been automated by build scripts in [9].

OpenCV has been around for a long time and therefore it has been thoroughly documented. The build steps for most platforms can be found from [10].

Release Process

There is no concrete documentation about the release procedure of the OpenCV library available, therefore we devised a description of it based on the source code, tags, releases, milestones as well as the issues and their corresponding discussions. The frequency of new releases is very low, there have been only 3 new releases in the past year [11]. There has been usage of milestones [12], but its usage is not consistent and some milestones do not contain a single issue. It seems as though the integrators are deciding whether to create a new release or not based on their experience with the system.

There have been 3 major releases for OpenCV: 1.X.X, 2.X.X and 3.X.X. As can be seen from the previous list, minor revision numbers have been used in addition to the major revision number. Alpha, Beta and RC (release candidate) revisions have been used for a new major version in order to slowly introduce the major releases and increase their reliability [13].

The OpenCV community uses GoogleTest to manage the testing of the library. A separate OpenCV bot is used to run the tests for each pull request, therefore reducing the workload of the developers. The testing has been standardized and there are instructions available on how to write tests.

We believe, that there should be at least a concise description of release testing and milestones should be used consistently. This would make it easier for the developers to decide whether to risk creating a private release or wait for a public release.

Configuration Management

The OpenCV community uses a single repository for all platforms. Furthermore, a single tag and a single release is made for every release, therefore the binaries for all platforms can be compiled from a single tag. The branches are platform-independent as well, since there are only 3 active branches: master, 2.4 and 2.4.12.x-prep [13].

This means, that the integrators compile different release versions (for multiple platforms) from the same source code and upload the new versions to their distribution web-page [11]. The process itself is not described in the documentation of OpenCV.

The Evolution Perspective

The evolution perspective discusses how the system has evolved during its lifetime, how flexible the system is when new functionality needs to be added and how maintainable it is.

The Evolution of OpenCV

OpenCV has been around for more than 17 years and has seen an enormous growth during this period. Not only has the number of algorithms increased, but there have been a lot of technological advancements, that have made new platforms important for computer vision applications. This includes the widespread use of Android and iOS as well as the introduction of hardware accelerators such as CUDA and SSE. These factors have forced the developers to keep up constant improvement, the results of which are presented in Table 3. This provides an overview of the major versions, how the folder structure has evolved as well as some of the new features introduced.

Version	Folder Structure	New Features & Refactorings
1.0.0	There is almost no common ground in comparison to the current version.	Very little functionality, uses Make.
2.0.0	The functional modules are grouped in a single *src* directory. A separate folder for third party libraries is created.	CMake is used instead of Make. No new features, mainly improvements.
2.1.0	A folder named *include* for the headers is created in the root directory.	No new features, mainly improvements.
2.2.0	The library has been reorganised and a *modules* folder is created.	*core*, *imgproc*, *features2d*, *objdetect*, *calib3d*, *ml*, *highgui*, *gpu*, *contrib* and *python* modules are created.
2.3.0	-	A new module, *stitching*, has been added. Also, there have been many smaller improvements to the modules
2.4.X	A separate *platforms* folder for operating system dependent scripts.	There are 12 minor releases, which add a lot of new algorithms and features, GPU functions and module refactorings. Added support for OpenCL, iOS and Android.
3.0.X	-	More GPU functions, improved Java bindings.
3.1.X	-	Support for newer operating systems and tools, a lot of new functionality.

Table 3: Evolution of OpenCV.

Software architecture sometimes has to be changed with the introduction of new features. This can be seen from OpenCV, which has seen a number of folder restructurings, module refactorings as well as minor refinements. This has ensured the longevity of the library, since it would be almost impossible by now to maintain it if these steps would have been skipped.

While contributions from the user community have played a valuable part of evolving the library, care must be taken when developers from all around are allowed to contribute. A few procedures that aim to reduce technical debt issues are maintained by the OpenCV integrators. A coding style guide is maintained and all contributors are urged to follow this guide. A similar guide on how to contribute is also presented on the OpenCV official site. Contributors should make sure the problem still persists before submitting something, and make sure that nobody else is working on the same issue, thus avoiding parallel development.

The Future of OpenCV

It is hard to predict the future, but it is most probable that OpenCV will continue to improve its performance, add new functionality and provide easier to use interfaces. A book on OpenCV [3] proposes six main areas, which will see major attention from the developers:

- *Applications* - There will be more higher-level applications, that can be easily used by the developers. This means lowering the learning curve and achieving faster time to market.
- *3D* - There will be more support for camera-sensor combinations, that enable the creation of 3D applications.
- *Dense Optical Flow* - There will be support for dense optical flows, which enable the detection of the movement of whole objects.
- *Features* - There will be a framework for interchangeable interest-point detection and interchangeable keys for interest-point identification.
- *Infrastructure* - The environment in which OpenCV is used will be improved, e.g. documentation improvements, better wrappers and better support for operating systems.
- *Camera Interface* - The handling of cameras will become easier and there will be support for more colors per channel.

Deployment View

The deployment view [4] focuses on the system's runtime environment, including the required hardware and mapping of software elements to the environment. OpenCV can be deployed e.g. on mobile devices using ARM processors running iOS, embedded systems running Linux or expensive workstations which include many CPUs and GPUs. Due to the extensive possibilities of deploying a system using the OpenCV library, we'll focus on one example which we'll use throughout this section. To cover all the possibilities is perhaps impossible, at least the material could easily fill a whole book.

The subject will be the computer vision part of an object tracking robot, running on Ubuntu using ROS (meta operating system designed for robotics) and GPU acceleration through CUDA. For simplicity the object detection is color based. Table 4 shows the technology dependencies for the system used in our example.

Component	**Requires**
Data capture	XIMEA xiD camera, XIMEA enabled
GPU acceleration	NVIDEA GeForce 770 graphics card, CUDA enabled
Data routing	ROS Indigo Igloo w/vision_opencv package
Meta-OS (ROS)	Ubuntu 14.04 LTS

Table 4: Technology dependency model for the example robot.

The hardware and OSs were chosen to represent a realistic case which shows how OpenCV can be used as a part of a system. ROS currently only has install packages for Ubuntu. Although ROS can be used to capture video from cameras, our example uses OpenCV modules for this task. Along with the hardware requirements listed in Table 4, the OpenCV library must be built with the XIMEA and CUDA options enabled to allow for communications with the camera and GPU acceleration.

The high-level operation of the robot is described in the following steps:

1. Get frame
2. Send frame to GPU
3. Detect object
4. Report results
5. Collect data from other sensors
6. Make decision based on results and data from sensors
7. Send control message to motor controllers based on decision
8. Goto step one

The video stream from the camera is captured using OpenCV's *cap_ximea* module. It is then converted to a GPU video object and sent to the GPU for processing. Sending the data to and from the GPU requires processing of both the CPU and the GPU. The image processing algorithm is handled solely by the GPU, allowing the CPU to focus on other tasks meanwhile, such as communications with other sensors and control calculations for the motors. Based on the results of the object detection, the central controller decides on an appropriate message to the motor control. Figure 4 provides a runtime platform model for the robot, with focus on the OpenCV related part.

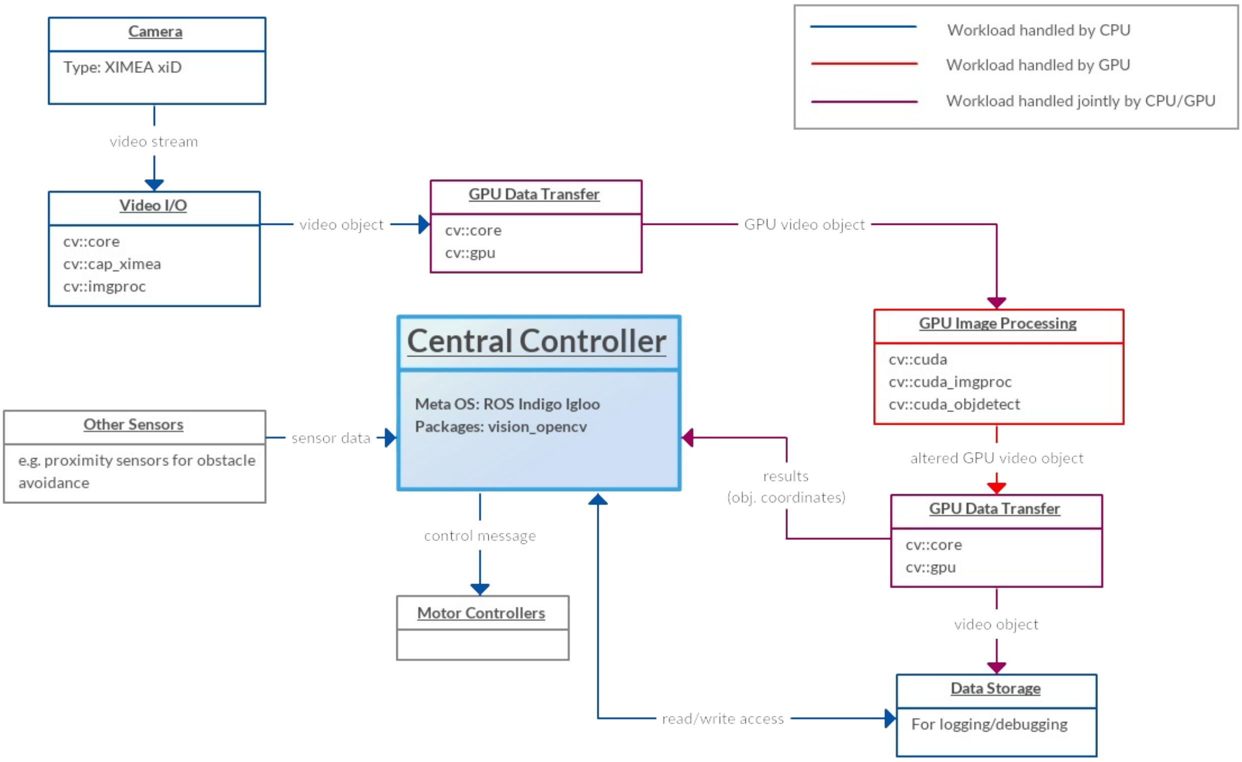


Figure 4: Runtime model for the example robot.

The Performance and Scalability Perspective

The Performance Perspective

Performance refers to the capability of a system to provide desired output within a certain response time. In image processing libraries such as OpenCV that deal with a large number of frames per second, it is favorable that the code is not only providing the correct solution, but also in the fastest manner possible.

OpenCV provides several features to boost performance and increase productivity. Many of these optimizations usually differ based on the host architecture, operating system and even the language being used. For instance certain optimization features can only be used on Windows running on an Intel chipset. Performance might even vary for certain functions depending on the language used. In the following section we discuss certain important optimization features provided by OpenCV. We also discuss methods to benchmark an application in OpenCV and important results obtained.

Default Performance Optimizations in OpenCV

Many of the OpenCV functions are optimized using different performance optimization features. It also contains unoptimized code. Exploiting these features is a possibility if the run-time system supports them. OpenCV runs the optimized code if it is enabled, else it runs the unoptimized code. The function `cv2.useOptimized()` can be used to check if it is enabled or disabled and `cv2.setUseOptimized()` to enable or disable it. Some of the important optimization features are listed in Table 5.

Feature	Description	Availability
SSE2/3	Streaming SIMD Extensions 2, is one of the Intel SIMD libraries. These instructions allow multiple data sets to be operated using a single instruction.	Intel chipsets
NEON	Specifically designed by ARM to accelerate multimedia and signal processing algorithms such as video encode/decode, 2D/3D graphics, gaming, audio and image processing.	ARMv6
IPP	Integrated Performance Primitives (Intel® IPP) is an extensive library of software functions for developing data processing and communications applications. These functions are highly optimized using Intel® Streaming SIMD Extensions.	OpenCV 3.0 on Intel Chipsets
OpenCL	Open Computing Language (OpenCL) is an open standard for writing code that runs across heterogeneous platforms including CPUs and GPUs. OpenCL provides applications with an access to GPUs for non-graphical computing (GPGPU) that results in significant speed-up.	All platforms
CUDA	CUDA is a parallel computing platform and application programming interface (API) model created by NVIDIA, to perform image processing on NVIDIA GPUs.	NVIDIA GPUs

Table 5: Performance enhancing features in OpenCV.

Some of the profiling results measured with CUDA show a significant improvement in performance for many OpenCV functions, as is depicted in Figure 5.

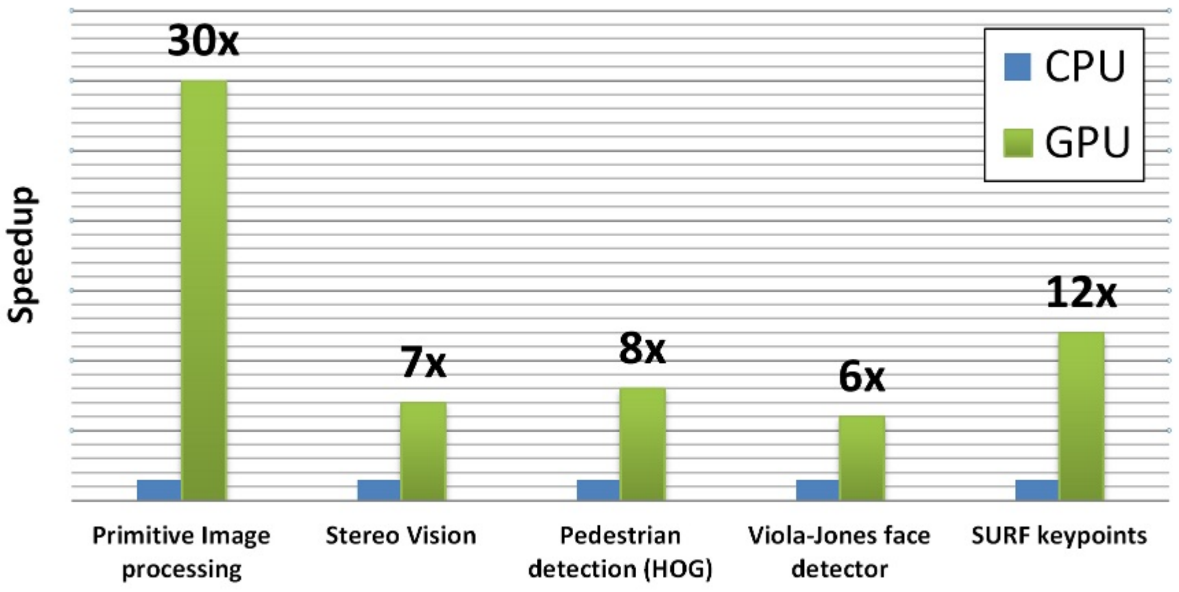


Figure 5: Performance enhancement of various OpenCV functions with CUDA (*Source: OpenCV official website*).

Measuring Performance with OpenCV

In OpenCV the performance is measured by counting the number of clock ticks between the start and the end of an OpenCV function. OpenCV provides API functions such as `cv2.getTickCount` which provides the number of ticks at a specific point in time. So if we call it before and after the function execution, we get an estimate of the number of clock-cycles used to execute a function.

The Scalability Perspective

Scalability of any software is its ability to handle increased workload in the future or for a short period of time, while continuing to meet the existing throughput, performance and responsiveness of the system. Also, any changes made to the system must ensure that the system’s capacity to handle the workload is not affected. Since OpenCV is a library which is used by other applications, its scalability indirectly influences the performance, response and throughput of the application using it.

To analyse the scalability of OpenCV, the usage of the library in real-time computer vision applications is considered. These applications are characterized by firm timing constraints, and handle varying amount of data at different points in their execution. Usually, these applications deal with the transformation of data from a still or video camera into either a decision or a new representation, which in this case is done using the APIs of OpenCV library.

The amount of data passed to the APIs (e.g. the size of the images or frames) affects their execution time and hence the response time of the whole system. The larger the data, the more processing time is required. However, the OpenCV algorithms are optimized and have been stabilized over the years to handle such increased workloads and to ensure good responsiveness and performance. Many of the OpenCV functions are optimized using SSE2, AVX etc, and are enabled by default while compiling. The function `cv2.useOptimized()` is used to check if they are enabled/disabled and the function `cv2.setUseOptimized()` is used to enable/disable them. If these optimizations are not enabled, the unoptimized code of the OpenCV will run. A common way of developing is to have optimizations disabled while debugging to avoid reducing the code readability.

The OpenCV GPU module supports NVIDIA GPUs in order to utilize the parallel computing capabilities of GPUs. However, the current version of OpenCV does not provide assistance in utilizing multiple GPUs, the algorithms can only use a single GPU. To achieve real-time performance while processing high quality videos, using a single GPU may not suffice. Therefore, the library provides APIs that allow the user to manually distribute the work between multiple GPUs by splitting the tasks into threads. This would help in accelerating the application during increased workload.

Similarly, OpenCV utilizes only one CPU core at a time. Multiple cores can be utilized by enabling multithreading using OpenMP (Open Multi-Processing) or TBB (Threading Building Blocks). These can be enabled by compiling the OpenCV library with `WITH_OPENMP` or `WITH_TBB` options enabled.

Currently, most of the APIs in OpenCV support only image formats using 8 bits per colour channel. If the applications using the library use newer cameras that can supply 10 or 12 bits per channel, the OpenCV APIs will fail to process the data. OpenCV, in the coming software releases, is planning to support more seamless handling of cameras along with eventual support for cameras with higher dynamic range. The higher bit depth increases the throughput of data to be processed by the algorithms.

Conclusions

We've made an effort to map the architecture of OpenCV to the ideas presented in [4]. This has proved to be a nontrivial task as it is hard to identify an architectural structure in the library. The interdependencies of the modules are numerous and hard to organize into categories. The size of the integrators group for the project is very small relative to the number of lines of code. The wide support for operating systems and hardware further increases the complexity of the system. A good portion of the bugs reported on the library's GitHub page are related to more recently added platform support, such as ARM processors.

Nevertheless OpenCV has done a good job during its lifespan making the world of computer vision and machine learning accessible to people, providing an easy to use interface to deal with complex tasks, without requiring the expert knowledge needed to implement the algorithms.

OpenCV provides excellent documentation and user guides for regular users who want to integrate OpenCV into their applications, but we feel that the documentation for the developers is scarce and therefore limits the amount of people willing to contribute to the project. Communication is slow and limited, which can also be attributed to the limited number of integrators.

References

1. About OpenCV. (2016). Retrieved February 29, 2016 from http://opencv.org/about.html
2. OpenCV. (2016). Retrieved February 29, 2016 from http://en.wikipedia.org/wiki/OpenCV
3. Bradski G. and Kaehler A. (2008). *Learning OpenCV*. O'Reilly Media, Inc.
4. Rozanski and Woods. (2012). *Software Systems Architecture*, 2nd edition. Pearson Education, Inc.
5. How OpenCV-Python Bindings Works?. (2015). Retrieved March 12, 2016 from http://docs.opencv.org/3.1.0/da/d49/tutorial_py_bindings_basics.html#gsc.tab=0
6. The OpenCV Coding Style Guide. (2015). Retrieved March 2, 2016 from https://github.com/Itseez/opencv/wiki/Coding_Style_Guide
7. Open Source Computer Vision Library. (2016). Retrieved March 2, 2016 from https://github.com/Itseez/opencv
8. Installation in Linux. (2016). Retrieved March 2, 2016 from http://docs.opencv.org/trunk/d7/d9f/tutorial_linux_install.html#gsc.tab=0
9. Platforms in OpenCV. (2016). Retrieved March 7, 2016 from https://github.com/Itseez/opencv/tree/master/platforms
10. Introduction to OpenCV. (2016). Retrieved March 7, 2016 from https://github.com/Itseez/opencv/tree/master/doc/tutorials/introduction
11. OpenCV Downloads. (2016). Retrieved March 12, 2016 from http://opencv.org/downloads.html
12. OpenCV Milestones. (2016). Retrieved March 12, 2016 from https://github.com/Itseez/opencv/milestones
13. OpenCV Tags. (2016). Retrieved March 12, 2016 from https://github.com/Itseez/opencv/tags
14. OpenCV Change Logs. (2016). Retrieved March 12, 2016 from http://code.opencv.org/projects/opencv/wiki/ChangeLog
15. Computer Vision. (2016). Retrieved March 22, 2016 from https://en.wikipedia.org/wiki/Computer_vision

OpenTripPlanner: A multimodal tripplanner

Marieke van der Tuin, Thom Hubers , Piotr Tekieli, Aafke Croockewit

Abstract

In this chapter, a multi-modal and multi-agency journey planner is analyzed: Opentripplanner (OTP). OTP is an open source platform, following a client-server model, providing several map-based web interfaces as well as a REST API for use by third-party applications. Users of OTP are most of the time developers of the project as well, since they use OTP to build applications and fix bugs themselves. In collaboration with one of the stakeholders of OTP, Plannerstack, contributions have been made to the project, all to add value to the 1.0.0 stable release, were developers are currently working on.

Table of Contents

Introduction

OpenTripPlanner (OTP) [4] is an open-source software for multimodal trip planning and transportation network analysis that allow users to search for itineraries including pedestrian, bike, transit, and car components. The project is used as routing engine in both commercial and open source projects.

This chapter informs about the recent development of OTP. First, the stakeholders are examined, followed by an analysis of different views on the system. These views provide insight into the inner workings of OTP itself and its development process. The insights gained with this analysis are combined into recommendations, which can be found at the end of this chapter.

History

OpenTripPlanner was created in 2009 by OpenPlans. The main user of the project was the TriMet [7] regional trip planner. TriMet [8] is the transport authority of Portland, USA. At the end of 2012, OpenPlans decided to stop with the development coordination. However, in the beginning of 2013 members of OpenPlans founded Conveyal, an open source focused transportation consultancy. They continued to support the OTP development.

Who is involved in OTP?

The history of the OpenTripPlanner [9] has a clear impact on the stakeholders involved [6]. The people involved mostly with the development of the code are employees of Conveyal [2]. Furthermore, a few local authorities that use OpenTripPlanner are actively involved in the project development. For example Ruter#, the travel agency for Southeastern Norway and HSL Finland use OpenTripPlanner in their planning tools. Not only local transport authoritie, but also some individuals are working with OpenTripPlanner. Most of them use OTP to built apps, such as AllyApp, a local transport planning application which can be used in multiple cities (mainly in Germany).

Users

There are two main types of people who use the system. There are the users who built an app or website using OTP (for example HSL Finland), and there are users that extend OTP into customer-based solutions (for example PlannerStack, who provided an OTP solution for the Dutch Connexxion bus company). The latter category can be seen as providers of OTP to others. Both groups can be further categorized into developers and benefiters. Developers contribute to the project by providing new features and bug fixes. Benefiters only make use of the system. The following graph shows the users and most important involved companies according to these two dimensions (*Figure 1*).

At last the end-users have to be mentioned. The end-users will use the app, website or other solution build on top of OTP, e.g. to plan a trip. Because this group of users is not directly involved in OTP (they will never directly use OTP), they are out of scope for this chapter.

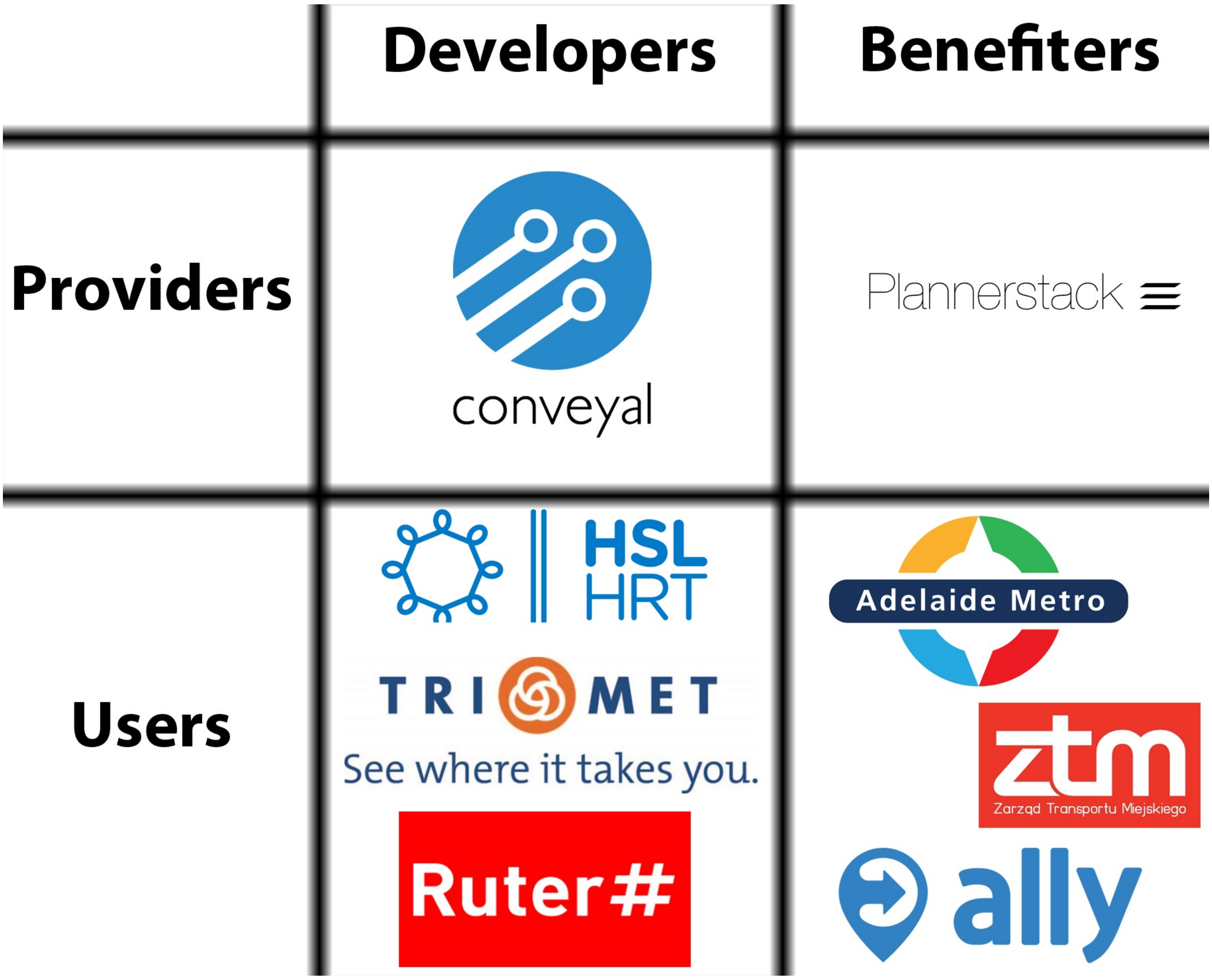


Figure 1: Categorization of the users

Contributors

It's hard to make a distinction between different types of contributors. There are no explicit testers, documentation writers or maintainers. Most contributors are involved through their company or an application. The main contributors are shown below with their profile pictures, names and GitHub user names (*Figure* 2). The people of the GitHub OpenTripPlanner organization are marked with a red square. The people shown at the bottom of the picture do not belong to a larger company, but all have made an app or website incorporating OTP.

pictures, names and GitHub user names (*Figure 2*). The people of the GitHub OpenTripPlanner organization are marked with a red square. The people shown at the bottom of the picture do not belong to a larger company, but all have made an app or website incorporating OTP.

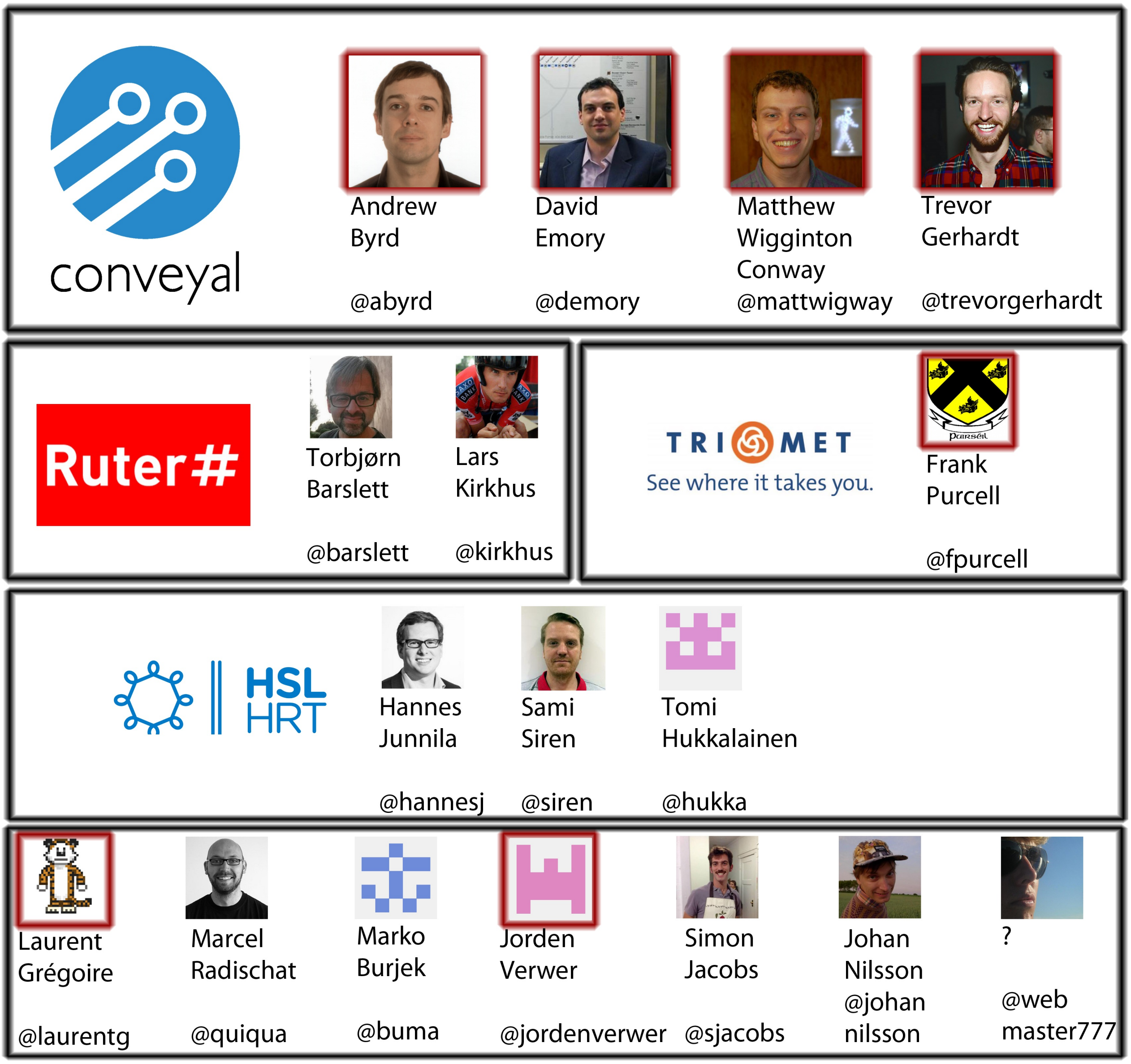


Figure 2: Overview of the main contributors, including their companies

Integrators: Andrew and Laurent

Andrew Byrd (@abyrd) is the overall manager of OTP and acts like an integrator. He decides which new features should be added and how bugs should be solved properly in order to remain the ideas behind OTP. On GitHub he is responsible for nearly all pull request merges. His role as a manager can clearly be distinguished at issue #2153. He shows clear understanding of the ideas behind OTP and pushes solutions directly to the codebase. At the Mailing Lists, he does not answer questions on the normal usage of OTP, but he does read through the discussions and tries to manage the resources correctly by for example redirecting discussions towards a GitHub issue if it would better belong there (for example at of some one else claiming it still does not work or a new bug appeared. Even compilation errors sneak through (see here)! It is remarkable that a manager of an open source project does not like to be controlled himself while requesting this from the other contributors.

Plannerstack

Plannerstack is one of the companies (actually a foundation) that is using OTP for it's projects. The developers of Plannerstack are improving OTP while using it. The foundation does also offer a hosted service in which developers can use the service without the need of a server.

Interests of contributors

As shown, the main developers are working for their own companies or applications. This also reflects the way developers maintain the code and add new features. Bugs are encountered by users of the system, and most of the time immediately fixed by themselves. For example, @sdjacobs reports an issue about a NullPointerException (#2210) and gives the fix himself immediately afterwards in a pull request (#2211).

New features do not seem to be added a lot the last month. This might be due to the upcoming release 1.0.0, the first stable release of the project. There are a few new features identified for this release though. An example of a new feature which is planned to be added is the Park and ride option. As can be read in issue #1330, this already has been implemented by the company GoAbout and only needs to be merged. We suppose that GoAbout needed this feature for their own project, and therefore implemented it. This seems to be the case for all new features added: they are implemented by contributors because their own application needs it. OTP can then benefit from those features built for others.

Context View

Its core is written in Java, therefore the application can be implemented on top of most of modern operating systems supporting JVM technology, including a variety of portable solutions. Each running instance of OTP offers a customizable web interface enabling efficient information exchange between the application's server and its users, mainly used for debugging purposes by developers. The provided API interface, on the other hand, allows more advanced users to automate their OTP implementations with scripts written with Jython or Groovy.

The development of OTP relies on the concept of open-source, both in terms of application's code, as well as implemented data standards. General Transit Feed Specification (GTFS) providing schedule data of public transportation, OpenStreetMap (OSM) delivering

The development of OTP relies on the concept of open-source, both in terms of application's code, as well as implemented data standards. General Transit Feed Specification (GTFS) providing schedule data of public transportation, OpenStreetMap (OSM) delivering information regarding street networks of particular zones, and NED tiles for assuring that the digital elevation model is applied to an active map, are the examples of such, which successfully substitute their commercial counterparts. The main distribution of application's source is realized by GitHub, which additionally allows an efficient publication and management of software related issues and associated pull requests, carrying potential changes to the project proposed by developers or community itself. The remaining project's documentation has recently been outsourced to an external website associated with project's main domain [5]. To allow better understanding of that concept, we have depicted these relations on the graph below (*Figure 3*).

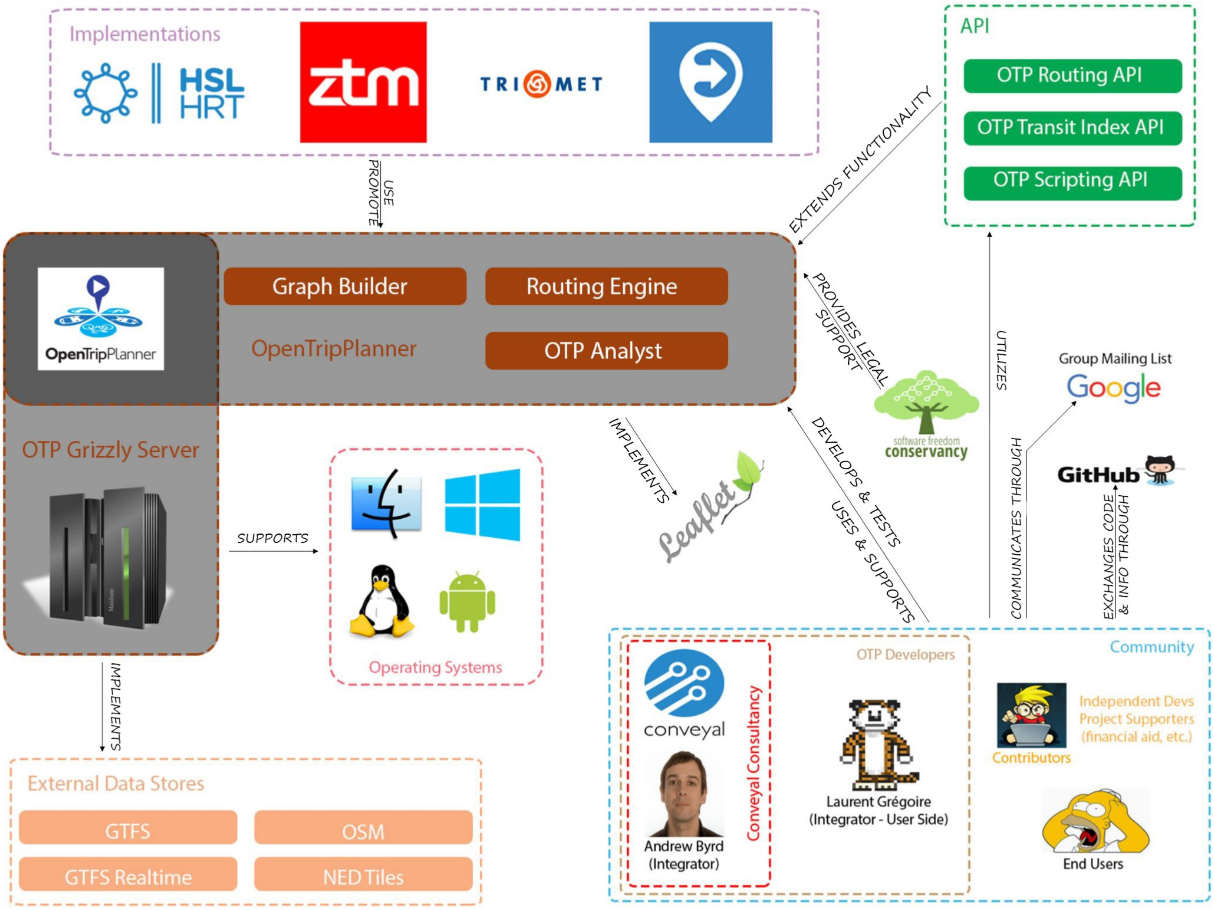


Figure 3: Context View of OpenTripPlanner

Development View

This document gives an insight into the development process and architectural structure of OpenTripPlanner (OTP). In the first chapter, a description of general development process can be found. Consequently, a development view is presented, and followed by some

Development of OpenTripPlanner

OpenTripPlanner uses Maven for building the project and managing its dependencies (the detailed information regarding them is located in the following paragraphs). It can be easily developed with the use of IDEs (Integrated Development Environment) such as Eclipse, IntelliJ or Netbeans. In case someone is willing to contribute to the project, the developer's mailing list and the GitHub repository can be used as a starting point for the process. OTP incorporates the aforementioned mailing list mainly for modification proposals and decision processes. GitHub, on the other hand, is used as the main code repository, place for publication of functionality issues and the maintenance of documentation. Although the guidelines specify that each commit should be based on a certain issue [10], this does not happen very often in practice. Most Pull Requests are correctly linked with the corresponding issue, but direct pushes of commits do not usually refer to them (defined issues) (e.g. this commit).

The project uses the same coding conventions for Java, as those that are included in GeoTools, which reflect a minor variation on the Sun coding convention. This convention is a way of formatting code and can automatically be set and utilized in any IDE which supports the import feature of the formatter.xml file (such as Eclipse). This file is positioned in the root folder of the project's repository. Furthermore, the development of JavaDoc is required for all new methods, classes and fields. All testing components are written with the use of JUnit.

The code itself is structured neatly. The codebase is split into four main parts: **client** which provides user's interface, **main** including all Java code for processing all requests and responses, the scripts which can be used to run OTP with a default set of parameters, and **tests** which provide the aforementioned testing modules for the main code. The utilization of of Java packages within **main** and **test** parts is correct and well justified, which reflects the sign of prior thorough planning and consideration of design decisions. Some packages include a separate package-info file, explaining what the package is responsible for and which other packages are related to it. The testing part reveals the exact package structure as the one spotted during the analysis of the core, which makes it easy to find the corresponding tests for certain classes and methods.

As-intended Development View

OpenTripPlanner operates in two modes: as an instance running via Grizzly server (web server), which is suited for planning trips, and an analyst mode in which the complete network can be analyzed. Both of them refer to the same core: a routing Java library which is responsible for computing the multi modal transport trips. On top of this core, several services are built which call its internal functions [11].

OpenTripPlanner operates in two modes: as an instance running via Grizzly server (web server), which is suited for planning trips, and an analyst mode in which the complete network can be analyzed. Both of them refer to the same core: a routing Java library which is responsible for computing the multi modal transport trips. On top of this core, several services are built which call its internal functions [11].

Here are presented the two of them that are used on a daily basis. The first one is Routing API, a RESTful (REST is Representational State Transfer) web service which responds to journey planning requests. This, with the an appropriate mapping library can form a friendly web GUI (Graphical User Interface) that provides basic planning features. The other one is Script API, that introduces users the ability to create automation mechanisms and functions for OTP with the use of their favorite scripting language. At that moment only a limited number of these is supported.

The Analyst can be executed in two configurations: the basic Analyst and the Batch Analyst. The basic Analyst Web Services provide basic network analysis results, while the Batch Processor handles more complex examination tasks. It allows an implementation of very open-ended configuration, which includes, for example, an information regarding terrain's population rates and therefore provides a powerful tool for visualizing how transport networks affect access to urban facilities. Both the Analyst and Batch Analyst do not make use of services, but rather directly call methods inside the core.

When it comes to the core itself, it incorporates two separate services for its proper functionality: OTP Transit Index API and Graph Builder. The Transit Index delivers information to the core, recovering them from GTFS files, which specify routes, stops and frequencies of all transit lines. The Graph Builder composes graphs from available OpenStreetMap (OSM) databases, which handle open source geographical data. These are created mostly by community and available to anyone, who is willing to utilize them.

Additionally, the Graph Builder can be invoked by OTP instance via Graph Visualizer. This component provides a basic interface for recovering edges and vertices's metadata.

A visualization of the as-intended development view has been composed according to the prior description and depicted on the graph located below (*Figure 4*).

Layering violations in OTP

Although OTP seems to have a clear layering approach among the packages, where the instance invokes core function utilizing several APIs for that purpose, this matter is sometimes violated.

The example of such violation can be observed while the OTP Instance directly calls the core to create a new RoutingRequest parameter configuration, which sets available modes and speed limits. We suspect that those are probably set before the actual execution of call, since different modes require different start-ups of the core itself. All other parameters are correctly set using the Routing API.

Furthermore, it would seem logical in the structure if the Analyst also used APIs to communicate with the core. However, this is not the case here, since the Analyst always directly invokes the core. What seems to be an even larger violation, is that the Router API uses constructs defined in Analyst, for example a PointSet (a group of destinations). A neater solution would involve creating a service which defines such constructs, which is called by both the Analyst and the Router API.

The identified violations are shown on a package dependency graph place below (*Figure 5*).

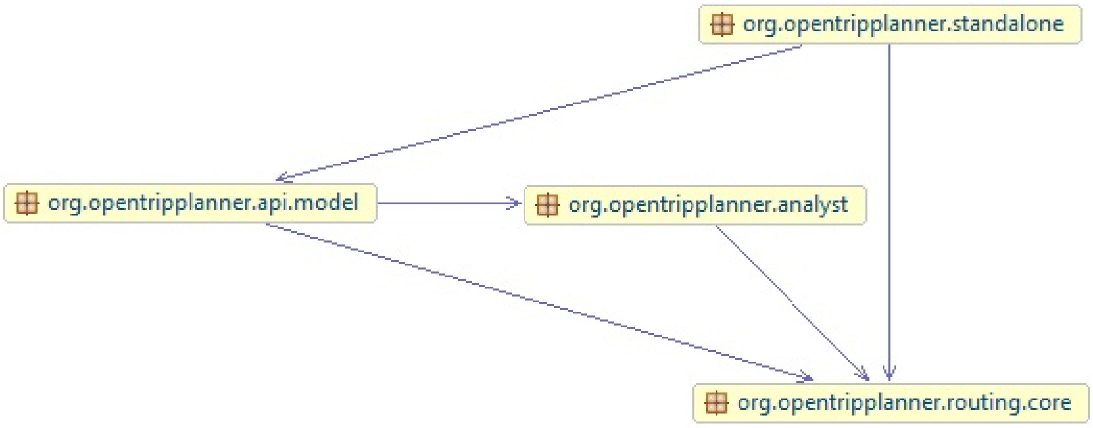

org.opentripplanner.standalone
org.opentripplanner.api.model
org.opentripplanner.analyst
org.opentripplanner.routing.core

Figure 5: Example of Layering violations in OTP

Technical debt in OTP

To identify if there is technical debt in the system, the OTP project is evaluated in terms of the following 7 points [12]: slowing velocity, stressful releases, aging libraries, defects, low automated test coverage, poor score on code quality metrics, presence of code smells

Slowing velocity

> Slowing velocity is the measurement for how much work a development team can complete during a particular interval. If the velocity starts to slow down, it can be a sign of a technical debt.

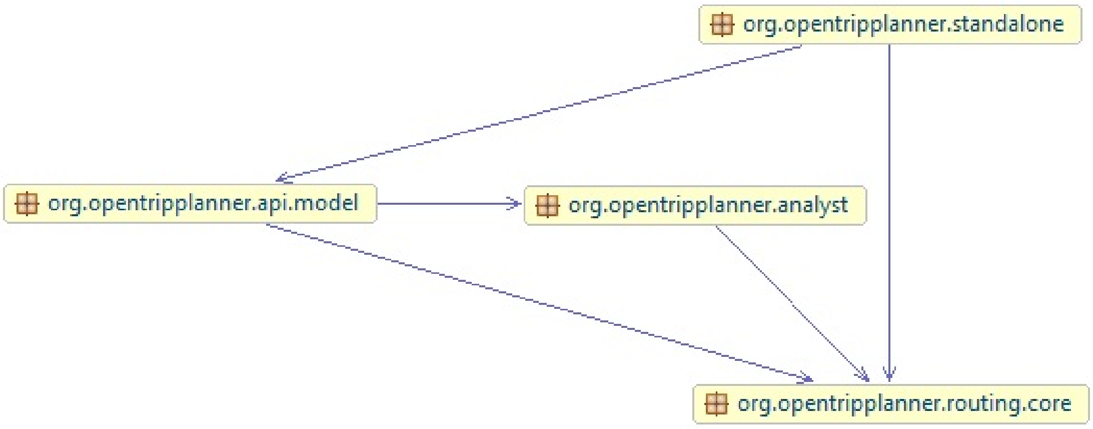

org.opentripplanner.standalone
org.opentripplanner.api.model
org.opentripplanner.analyst
org.opentripplanner.routing.core

Figure 5: Example of Layering violations in OTP

Technical debt in OTP

To identify if there is technical debt in the system, the OTP project is evaluated in terms of the following 7 points [12]: slowing velocity, stressful releases, aging libraries, defects, low automated test coverage, poor score on code quality metrics, presence of code smells

Slowing velocity

> Slowing velocity is the measurement for how much work a development team can complete during a particular interval. If the velocity starts to slow down, it can be a sign of a technical debt.

Since the beginning of 2016 there are not many contributions done to the master branch. Although, for establishing the conclusion that inactivity could be a sign of technical debt here, is probably too soon. To justify that, we would like to mention that in 2015 the project was provided with many contributions. Besides, the same situation happened at the end of 2014, after which the project became active again. Additionally, about 10 PRs have already been merged to the master this month. All of them were provided by several different developers.

Stressful releases

> The stress at the end of a release cycle indicates an occurrence of a possible technical debt, which impairs the development productivity.

On Google Groups and docs.opentripplanner there is clear communication about new releases. The releases seem to be strictly controlled (in a good manner) mostly by Andrew Byrd, who takes the responsibility for these actions. The goals that are set are concrete and realistic, which prevents against slipping into crunch mode [13].

Aging libraries

occur not only at the beginning and end of a trip, but also at transfers. Many months have been spent on ensuring this requirement is met, and it has interfered with the implementation of new routing algorithms more than once. This clearly shows that some knowledge was not there yet in earlier phases of the project. In this case OTP should deal with bike transit. This is fitting what Ward Cunningham says about technical debt: “I am in favor of writing code to reflect your current understanding of a problem even if that understanding is partial. Technical debt is a knowledge deficit."

It took a long time (“..many months have been spent..”) to implement the improvement, according to AByrd on Google Groups.

Test coverage

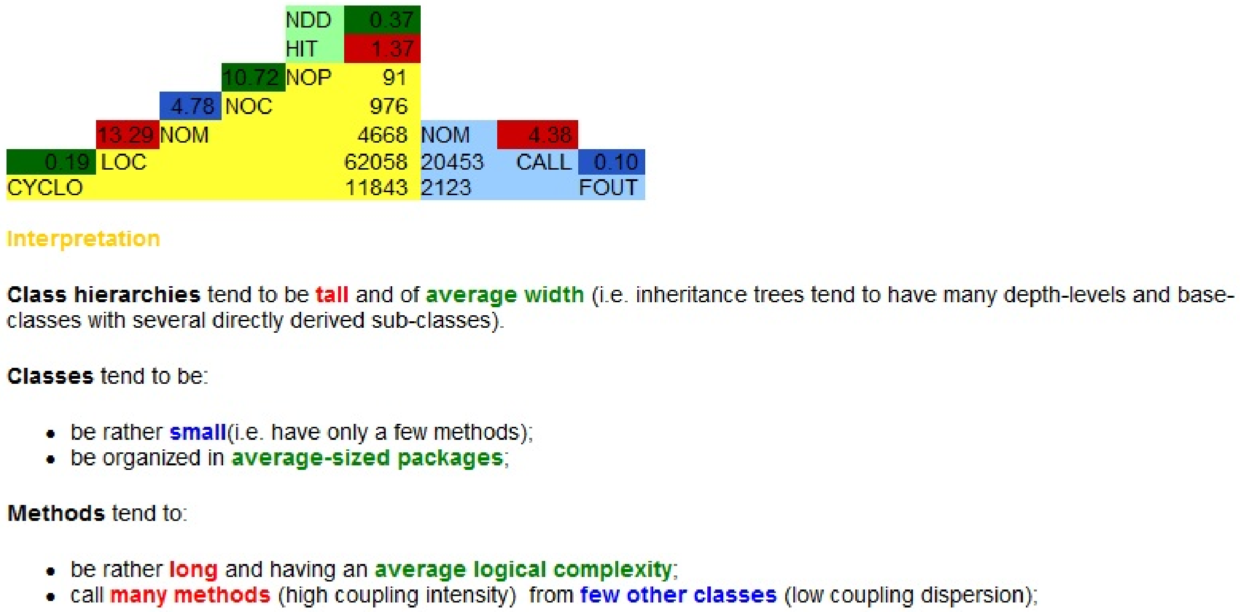
> The test coverage of OpenTripPlanner was tested using the tool EclEmma.

The overall test coverage of OpenTripPlanner is 55%, which is quite low. But the Routing package which contains all routing algorithms, has a test coverage of 75%, which is pretty good. The other main packages used by the core of OpenTripPlanner, GTFS and OpenStreetMaps, have a coverage of 84%. The test coverage for OTP as a whole is good enough.

OpenTripPlanner uses a continuous integration sever. Every time a Pull Request is submitted or a change pushed, this server will compile and test the new code. In this way, the code is continuously tested and not accidentally broken by a new commit. Therefore, concerning the testing no technical debt occurs.

Poor score on code quality metrics

The left side of the pyramid (*refer to figure 6*) shows the size and complexity of the system, showing a high rate for the number of methods. This means there is room for improvement for the number of methods and for the length of the methods.



Interpretation

Class hierarchies tend to be **tall** and of **average width** (i.e. inheritance trees tend to have many depth-levels and base-classes with several directly derived sub-classes).

Classes tend to be:

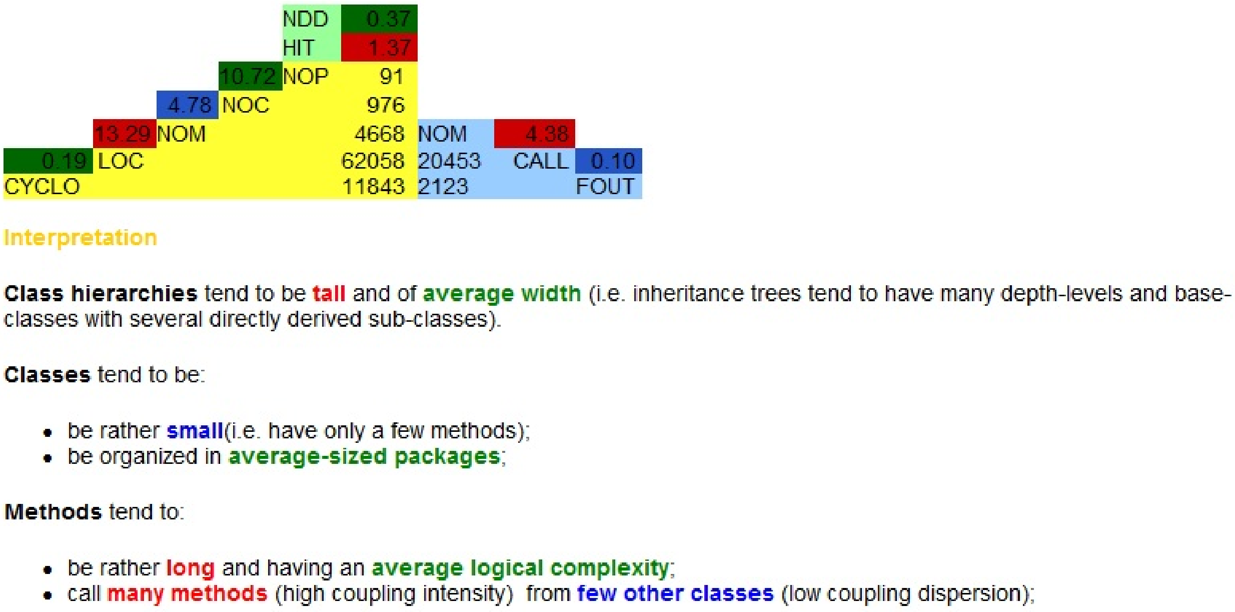
- be rather **small**(i.e. have only a few methods);
- be organized in **average-sized packages**;

Methods tend to:

- be rather **long** and having an **average logical complexity**;
- call **many methods** (high coupling intensity) from **few other classes** (low coupling dispersion);

OpenTripPlanner uses a continuous integration sever. Every time a Pull Request is submitted or a change pushed, this server will compile and test the new code. In this way, the code is continuously tested and not accidentally broken by a new commit. Therefore, concerning the testing no technical debt occurs.

Poor score on code quality metrics
The left side of the pyramid (*refer to figure 6*) shows the size and complexity of the system, showing a high rate for the number of methods. This means there is room for improvement for the number of methods and for the length of the methods.



NDD 0.37
HIT 1.37
10.72 NOP 91
4.78 NOC 976
13.29 NOM 4668 NOM 4.38
0.19 LOC 62058 20453 CALL 0.10
CYCLO 11843 2123 FOUT

Interpretation

Class hierarchies tend to be **tall** and of **average width** (i.e. inheritance trees tend to have many depth-levels and base-classes with several directly derived sub-classes).

Classes tend to be:

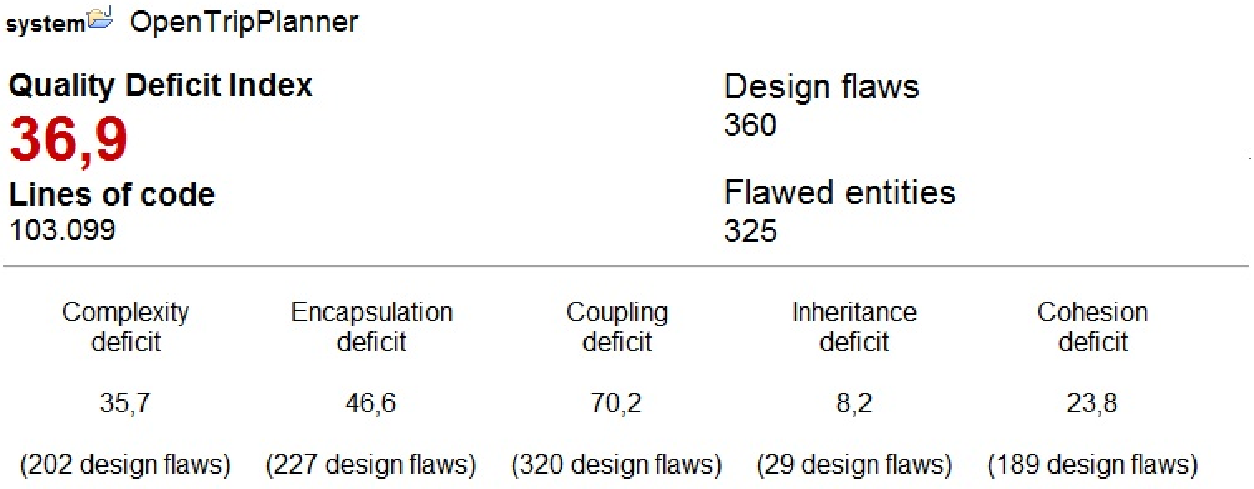
- be rather **small**(i.e. have only a few methods);
- be organized in **average-sized packages**;

Methods tend to:

- be rather **long** and having an **average logical complexity**;
- call **many methods** (high coupling intensity) from **few other classes** (low coupling dispersion);

Figure 6: Code quality of OTP

Presence of code smells** With Infusion Metrics the code smell is analyzed. OTP has 133.099 lines of code. The system has 360 Design Flaws and the QDI is 36,9 (*see figure 7*). (The Quality Deficit Index is a positive, upwards unbound value, which is a measure of "badness" of the analyzed system's design quality respecting the overall size of the system.)



system OpenTripPlanner

Quality Deficit Index
36,9

Lines of code
103.099

Design flaws
360

Flawed entities
325

Complexity deficit	Encapsulation deficit	Coupling deficit	Inheritance deficit	Cohesion deficit
35,7	46,6	70,2	8,2	23,8
(202 design flaws)	(227 design flaws)	(320 design flaws)	(29 design flaws)	(189 design flaws)

Figure 7: Presence of code smells

2. Graph build features

The Graph build features require a complete graph rebuild if they are changed. They can be specified in `build-config.json`. They specify options which should not be changed by the user of the application itself, for example transit fares. Application builders or transport authorities can use the Graph build features to change certain parts - for example a change in the fares - without any need to restart the complete application. This is useful since a restart of the application might require quite some time due to the time consuming loading of data. Typically, this will be done a few times per year in a running application.

3. Run-time features

A geographical area with specific services is called a router. OpenTripPlanner can deal with several routers at the same time. The configurations can be set per router, at run-time. These can be specified in a separate `router-config.json` file, but most of the time the user interface will provide options for changing them. The run-time features include user specific settings, such as modes to use and maximum walking distance.

Relationships

Several relationships can be found between the features.

- If a certain **Mode** is not selected, the accompanied options are not taken into account, such as **Fares**, **Speeds** and **Boarding time**.
- If cycling is added as a selected **Mode**, walking will be added automatically as well, with a high **Reluctance** for walking. This was implemented to model walking next to your bike, which might be needed to reach a walk-only accessible area.
- If **Wheelchair accessibility** is selected, non-wheelchair accessible **Mode** settings will not be used, for example cycling.
- If the **Visualizer** is started, the options which are set in the JSON files will not be used. Part of the options can be selected in the visualizer itself by selecting checkboxes in the user interface, as can be seen below (*figure 8*).

Relationships

Several relationships can be found between the features.

- If a certain **Mode** is not selected, the accompanied options are not taken into account, such as **Fares**, **Speeds** and **Boarding time**.
- If cycling is added as a selected **Mode**, walking will be added automatically as well, with a high **Reluctance** for walking. This was implemented to model walking next to your bike, which might be needed to reach a walk-only accessible area.
- If **Wheelchair accessibility** is selected, non-wheelchair accessible **Mode** settings will not be used, for example cycling.
- If the **Visualizer** is started, the options which are set in the JSON files will not be used. Part of the options can be selected in the visualizer itself by selecting checkboxes in the user interface, as can be seen below (*figure 8*).

As it can be observed, the before mentioned list has been created and divided accordingly to the binding times of specific features. In this particular software, most of them can be selected or altered during run-time with the use of GUI or CLI commands. Those which cannot be configured that way are usually defined as starting parameters for each OTP instance. The analysis of source code has not revealed any compile-time bounds or features that could personalize each build. The graph located below (*figure 9*) depicts a simplified view on the structure of listed features along with their dependencies and logical constraints. The reason for this simplification lies within the complexity that is introduced with each presentations of detailed architectures. In order to make this graph feasible for further analysis, and understandable to each stakeholder, this level has been set to be optimal, meaning - without additional elements that has not been described in the prior parts.

As it can be observed, the before mentioned list has been created and divided accordingly to the binding times of specific features. In this particular software, most of them can be selected or altered during run-time with the use of GUI or CLI commands. Those which cannot be configured that way are usually defined as starting parameters for each OTP instance. The analysis of source code has not revealed any compile-time bounds or features that could personalize each build. The graph located below (*figure 9*) depicts a simplified view on the structure of listed features along with their dependencies and logical constraints. The reason for this simplification lies within the complexity that is introduced with each presentations of detailed architectures. In order to make this graph feasible for further analysis, and understandable to each stakeholder, this level has been set to be optimal, meaning - without additional elements that has not been described in the prior parts.

As can be read above, OTP is not very variable. There are a lot of options which influence the routing mechanism, but it's hard to see those as different software versions, since none of them are configured at compile-time or can personalize a certain build. There are some different operational modes: normal, script and analyst. These do all provide a different subset of the features of OTP and can thus be seen as variants. But those different modes are never being shipped separately and a big part of the code base is being used by all modes.

The strategy to cope with variability can be summarized as trying to have as less variability as possible. The variables (routing parameters) described above are managed by configuration files. These files are formatted in JSON, which makes the configuration structured and easy readable by both humans and computers. The software reads these files and applies the settings on its working. This mechanism makes the variability manageable, because the source code doesn't need to be adjusted for each variety in functionality. The fact that the configuration files are shared across the different operation modes does also improve the manageability of the different features.

One exception to this solution however, is the way that options are covered in the Graph Visualizer. The visualizer has its own configuration section with a subset of the options of OTP. That this fragmentation reduces manageability can directly be seen, since the layout of this panel is broken since a recent update.

Evolution history of variability mechanism

The way variability is handled within the OTP project has not been changed so much in the last years. Configuring route planning software with configuration files is a common way to do this and the people who built the software chose this mechanism at the beginning of the development. Since it's working very well and ensures a good manageability, this mechanism is not likely to change a lot in the coming versions. A good next step might be that the current approach is used by the whole system and that exceptions like the Graph Visualizer will be removed. This improves clarity and manageability.

Handling of itinerary requests

Dealing with itinerary requests is the main functionality of OpenTripPlanner. To give more insight on the internal handling of these requests, a functional view was created for this purpose (*see figure 10*).

As can be read above, OTP is not very variable. There are a lot of options which influence the routing mechanism, but it's hard to see those as different software versions, since none of them are configured at compile-time or can personalize a certain build. There are some different operational modes: normal, script and analyst. These do all provide a different subset of the features of OTP and can thus be seen as variants. But those different modes are never being shipped separately and a big part of the code base is being used by all modes.

The strategy to cope with variability can be summarized as trying to have as less variability as possible. The variables (routing parameters) described above are managed by configuration files. These files are formatted in JSON, which makes the configuration structured and easy readable by both humans and computers. The software reads these files and applies the settings on its working. This mechanism makes the variability manageable, because the source code doesn't need to be adjusted for each variety in functionality. The fact that the configuration files are shared across the different operation modes does also improve the manageability of the different features.

One exception to this solution however, is the way that options are covered in the Graph Visualizer. The visualizer has its own configuration section with a subset of the options of OTP. That this fragmentation reduces manageability can directly be seen, since the layout of this panel is broken since a recent update.

Evolution history of variability mechanism

The way variability is handled within the OTP project has not been changed so much in the last years. Configuring route planning software with configuration files is a common way to do this and the people who built the software chose this mechanism at the beginning of the development. Since it's working very well and ensures a good manageability, this mechanism is not likely to change a lot in the coming versions. A good next step might be that the current approach is used by the whole system and that exceptions like the Graph Visualizer will be removed. This improves clarity and manageability.

Handling of itinerary requests

Dealing with itinerary requests is the main functionality of OpenTripPlanner. To give more insight on the internal handling of these requests, a functional view was created for this purpose (*see figure 10*).

as well as the graph and transit feeds. If requested, the real-time feeds for transit and historical data on traffic speeds is included in the Routing Context as well. The Context is then given to the PathFinder. The PathFinder is responsible for actually searching possible routes from origin to destination. This is done by applying the A star algorithm. Several heuristics are used in this algorithm to speed up, depending on the type of request. This leads to a ShortestPathTree with the discovered itineraries. From this list the itineraries are taken that actually arrive at the wanted destination, also called States. These are banned by the PathFinder from the considered paths for the next searches, to create more variety in the found routes. The PathFinder continues iterating until a certain time limit is reached or the number of paths found is sufficient. Next, the found paths are returned to the user.

OTP going international

The rapidly growing interest in digitalized solutions, including software and hardware, can nowadays be observed all around the globe. One of the keys to such a product's success lies within its ability to perform efficient communication with its end-users through simplified and localized interfaces. Translation is a very important factor, appealing to potential customers to buy the product or application, once it tries to enter and win the market. Despite being an open source project, the OTP is equipped with a convenient and easy mechanism for providing support of new languages. Our analysis has shown that a couple of localizations already exist, in most cases done by contributors from Europe. Following this trend, decided was to include the support of one, new language - Polish - which is designed to become a decision-driving factor for companies which originate in that country, but still hesitate on whether to choose this product or switch to commercial competitors.

The translation process of the web interface was conducted in accordance to instructions included in an official guide written by OTP developers. The first step was connected with re-using a dedicated translation template with English phrases as a base for new localization files. This was done with the help of POEdit software, which is a good solution for the manipulation of gettext (i18n) content. Since the software uses JSON structures for its provision of support for various languages, the files were converted and saved in the desired format. The last step involved the adjustment of the representation of units, and date / time formats, that were the most suitable for the newly generated localization. The graphical representation of the following steps in this process is included below.

as well as the graph and transit feeds. If requested, the real-time feeds for transit and historical data on traffic speeds is included in the Routing Context as well. The Context is then given to the PathFinder. The PathFinder is responsible for actually searching possible routes from origin to destination. This is done by applying the A star algorithm. Several heuristics are used in this algorithm to speed up, depending on the type of request. This leads to a ShortestPathTree with the discovered itineraries. From this list the itineraries are taken that actually arrive at the wanted destination, also called States. These are banned by the PathFinder from the considered paths for the next searches, to create more variety in the found routes. The PathFinder continues iterating until a certain time limit is reached or the number of paths found is sufficient. Next, the found paths are returned to the user.

OTP going international

The rapidly growing interest in digitalized solutions, including software and hardware, can nowadays be observed all around the globe. One of the keys to such a product's success lies within its ability to perform efficient communication with its end-users through simplified and localized interfaces. Translation is a very important factor, appealing to potential customers to buy the product or application, once it tries to enter and win the market. Despite being an open source project, the OTP is equipped with a convenient and easy mechanism for providing support of new languages. Our analysis has shown that a couple of localizations already exist, in most cases done by contributors from Europe. Following this trend, decided was to include the support of one, new language - Polish - which is designed to become a decision-driving factor for companies which originate in that country, but still hesitate on whether to choose this product or switch to commercial competitors.

The translation process of the web interface was conducted in accordance to instructions included in an official guide written by OTP developers. The first step was connected with re-using a dedicated translation template with English phrases as a base for new localization files. This was done with the help of POEdit software, which is a good solution for the manipulation of gettext (i18n) content. Since the software uses JSON structures for its provision of support for various languages, the files were converted and saved in the desired format. The last step involved the adjustment of the representation of units, and date / time formats, that were the most suitable for the newly generated localization. The graphical representation of the following steps in this process is included below.

OTP is an interesting project because it’s both commercial and open source. This means that there is a community that is involved in maintaining and improving the code, but this community is small at the same time. The authors are not very clear about their intentions regarding the future of the software. From a technical point of view, the project works quite well. Although the routing engine has some bugs, none of these are show stoppers and the project is used in some important trip planners.

If the project owners want OpenTripPlanner to become a lively open source project, it is recommended that they are more open about their goals for the future and involve the community in discussions about release schedules and features. Direct undiscussed commits to the master branch are not recommended. But it’s hard to keep a community involved and deliver software to clients at the same time.

References

1. Rozanski, N. & Woods, E. (2012). *Software systems architecture : working with stakeholders using viewpoints and perspectives.* Upper Saddle River, NJ: Addison-Wesley.
2. Conveyal Consultancy Official Website 'http://conveyal.com'
3. - Google Groups Mailing Lists 'https://groups.google.com/forum/#!forum/opentripplanner-users'
4. OTP's Official Website, 'http://docs.opentripplanner.org/en/latest/'
5. OTP's Documentation, 'http://docs.opentripplanner.org/en/latest/Deployments/#opentripplanner-deployments-worldwide'
6. Software Freedom Consservancy (SFC) Official Website 'http://sfconservancy.org/'
7. Trimet Trip Planner (https://ride.trimet.org/)
8. Trip Planner by Municipal Transport Company of Valencia 'http://www.emtvalencia.es/geoportal/?lang=en_otp'
9. http://www.opentripplanner.org/blog/2013/11/22/otp-joins-sfc/
10. http://docs.opentripplanner.org/en/latest/Developers-Guide/#contributing-to-the-project
11. http://docs.opentripplanner.org/en/latest/#basic-otp-architecture
12. https://18f.gsa.gov/2015/10/05/managing-technical-debt/
13. http://chadfowler.com/blog/2014/01/22/the-crunch-mode-antipattern/
14. http://verraes.net/2013/07/managed-technical-debt/

Ruby on Rails - Always on Track

Valentine Mairet
ValMai

Daniël Vliegenthart
vliegenthart

Aaron Ang
aaronang

Bastiaan van IJzendoorn
basvijzendoorn

Abstract

Ruby on Rails is a web framework, focusing on making the development of web applications easier and more fun. This framework has over 3,000 contributors, 700,000 users, and 200,000 lines of code. To help developers understand Rails, we first analyse the stakeholders of Rails and provide an overview of where Rails stands in the real world. We then describe Rails from various viewpoints and perspectives to understand its software architecture. Providing these insights should help people join the Rails community and contribute to Rails, and give Rails members a description of their system from an outsider's perspective.

Introduction

Rails is a Ruby framework designed to facilitate web development and to develop database-backed web applications. The Rails project was started back in 2004, and the first release of Rails occurred in December 2005. In 2008, Rails migrated to GitHub. The purpose of the

Rails framework is to make the development of web applications easier and before all, more fun [12].

As of today, there are over 700,000 websites built with Rails [1], 3,000 contributors to the Rails project, and this framework contains overall 200,000 lines of code. From seamless database integration to providing a REST API; from emailing support to rendering HTML with embedded Ruby code - Rails contains anything a web developer needs.

The Rails core development team meticulously relies on two concepts: Convention Over Configuration and Don’t Repeat Yourself. These software paradigms ensure the uniformity of the code base and the simplicity of development, without losing flexibility.

For the Software Architecture 2016 course, we chose Rails to investigate and to contribute to. For our contributions to Rails, we approached the core developers and they gave us a task to help them for the next release of Rais. It was an honour to assist them, and we got offered to work on a separate, though related, Rails project.

Concerning our research upon Rails, we report our findings in this chapter. We start off a with a stakeholder analysis to inquire upon who and which entities have an interest in the realisation of the framework. Then, we put Rails into context by showing where it stands in the real world. We then approach the functionalities of Rails and describe the modules that make up the framework. We investigate the variability of the framework, i.e. how Rails facilitates configurability.

Additionally, as security is an important aspect of the Web, we investigate how Rails deals with important security concerns. Finally, we conclude this report with a discussion of Rails nowadays.

Stakeholder Analysis

When it comes to designing a product for usage, one must think of who will have an interest in this particular product, even when we speak of software. These entities or people who hold a stake in the realisation of the product are called stakeholders. Rails has many stakeholders that we have identified in the table below. We have sorted them according to the eleven types proposed by Rozanski and Woods [18].

Type	Stakeholder	Description
Developers	Core developers, committer team and contributors	The core developers sets policies and are the general managers, the committer team helps the core developers with pull requests.
Acquirers	Core developers	The core developers decide on the overall direction of Rails. There are no business sponsors for Rails.
Assessors	Developers in general	Developers assess the conformance to standards and legal regulations themselves.
Communicators	Teachers	Rails is taught by **teachers** in online environments by means of instructive movies on YouTube, tutorials on the web, or answering questions on StackOverflow. There is a special project called “Ruby on Rails Guides” [16], which provides guidelines “designed to make you immediately productive with Rails”.
Maintainers	Core developers, contributors	The overall evolution is maintained by the core developers, all other maintenance tasks by contributors.
Product Engineers	Users, developers	The users are the engineers of Rails products, the developers manage the GitHub repository and the Rails information website.
Suppliers	GitHub	GitHub is a supplier for the Rails repository.
Support Staff	Developers, teachers	Support for the development of Rails is done by developers in mailing lists and on GitHub. The teachers provide support for Rails applications on platforms such as StackOverflow.
System Administrators	Core developers, committer team	They control the development of the project.
Testers	Core developers, committer team	They are responsible for the testing of new commits.
Users	Developers and organisations that use Ruby on Rails	Examples of projects that are built with Rails are: Basecamp, GitHub, Shopify, Airbnb, Twitch, SoundCloud, and many more.

Additionally, the following stakeholders are identified but do not fall within the types of Rozanski and Woods.

Gem Contributors: Some developers develop gems that can contain anything from libraries to self-contained applications that are useful for a Rails application.

End-Users: End-users are the people who view and use Rails website as consumers. For instance, a GitHub visitor.

RailConf Organisers: There is a yearly Rails conference in order to meet people of the Rails community in real life.

Bloggers: a blog is maintained by some developers that share news on the progress of Rails development.

Open Source Scientists: The TU Delft Software Architecture class, and other academics interested in open source, perform research on the Rails community and the Rails repository.

Ruby: Rails is written in the Ruby programming language. There is a cyclic dependency between the language and the framework, hence making Ruby a stakeholder of Rails.

Context Viewpoint

In this section, we describe the context model of Rails. This model describes what the system does and does not do and defines the relationships, dependencies, and interactions between the framework and its environment [18].

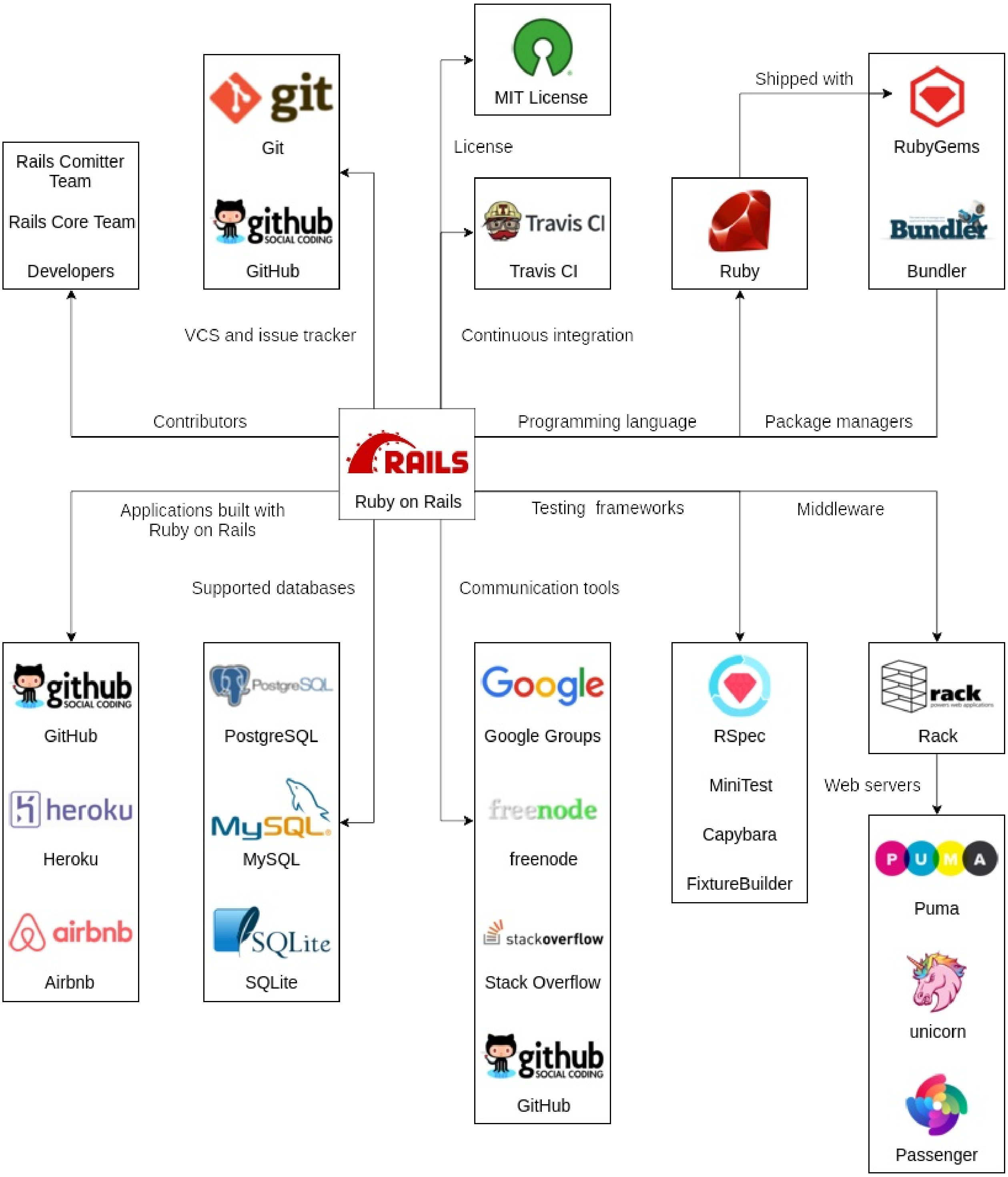


Figure 1: Context model of Ruby on Rails.

In Figure 1, we display the context model of Rails. For external entities that are not self-evident, we explain its role with respect to Rails.

VCS and issue tracker: GitHub and Git facilitate collaboration between developers using the pull-based development model. Furthermore, GitHub is used as an issue tracker.

Package managers: Ruby comes with two package managers: RubyGems and Bundler. On the one hand, RubyGems is shipped with Ruby and is normally used for installing packages, known as gems, on system level. On the other hand, Bundler is usually used to install dependencies based on a `Gemfile` which is included in almost every Ruby project. Note that Bundler itself is a gem, which can be installed using RubyGems.

Communication tools: The Ruby on Rails community is heavily active and makes use of different communication channels: Google Groups, Freenode, Stack Overflow, and GitHub. Google Groups is used to discuss the core functionality of the framework. GitHub serves as a mean to discuss issues, primarily bugs and perform code reviews.

Testing frameworks: As every application is developed to satisfy users' need, it is important that we can verify and validate its functionality. To facilitate testing, the Ruby community developed many testing frameworks each with its own target domain.

Supported databases: As more complex applications are developed, it is likely that one is required to store data. This is where databases play an important role. Databases provide means to read and write data to a storage device.

Middleware: *Rack provides a minimal interface between web servers that support Ruby and Ruby frameworks* [7]. It specifies a communication protocol between the web server and a Ruby framework, such that the Ruby framework can work with an HTTP request, and that the web server can construct a valid HTTP response from the Ruby framework response [2].

Web servers: A web server processes HTTP requests from its users using network protocols. Therefore, to make an application accessible from the Web, one will need to deploy a Rails application using Rack and a web server.

Functional Viewpoint

The functional viewpoint describes the system’s runtime functional elements and their responsibilities, interfaces, and primary interactions [18]. The functionalities are modeled in the UML diagram of Figure 2 and further described in this paragraph. Also, a common development scenario is described. The functionalities were extracted from the Ruby on Rails guides [16].

Functionalities

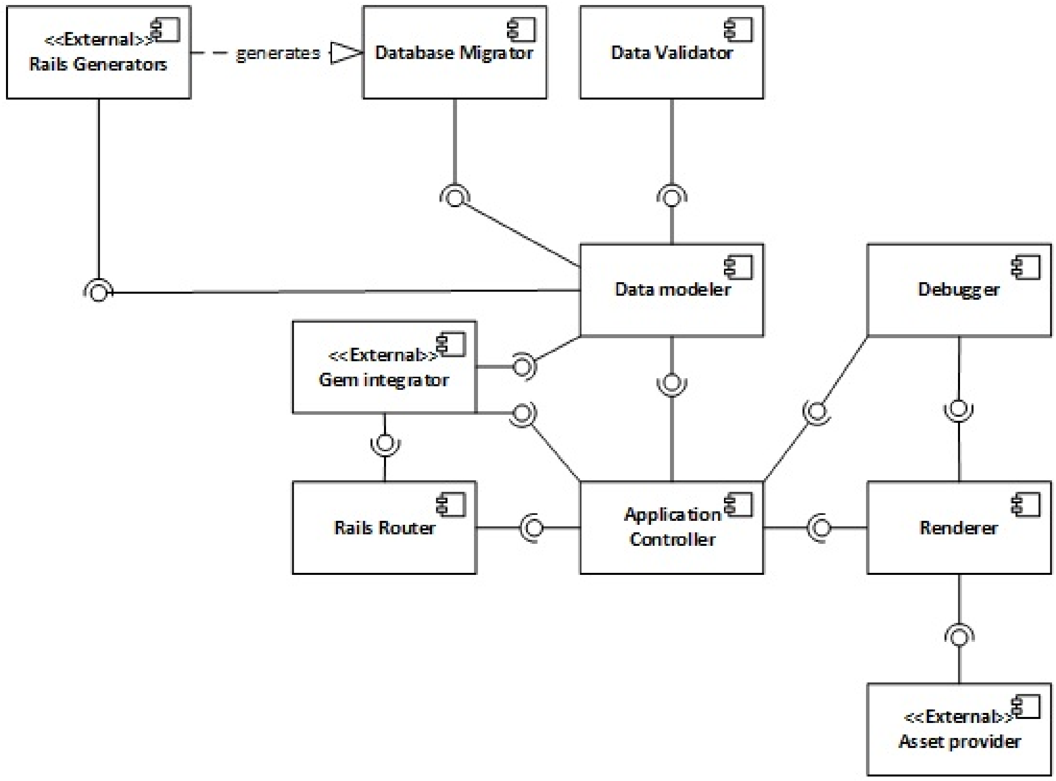


Figure 2: Functional structure model of Rails.

Rails Router

The Rails router has the responsibility to receive HTTP requests and to recognize URLs and dispatches them to a controller’s action. For each URL path, a controller and action is defined. When an HTTP request received , the router determines which controller and action to use. Then, an instance of the controller is made and the action method is called.

Application Controller

The application controller is responsible for manipulating the business logic of the application, initiating the create, read, update, delete (CRUD) operations, and the rendering process. Controllers can be defined and are interfaced by subclasses of the `ApplicationController`. The methods of the subclass are the actions of the controller.

Data Modeler

The data modeler has the following responsibilities:

1. Represent models and their data.
2. Represent associations between these models.
3. Represent inheritance hierarchies through related models.

4. Perform database operations in an object-oriented fashion.

Models are interfaced by subclasses of the `ActiveRecord::Base` class. The attributes of the class are mapped to the columns of the model table.

Data Validator

The data validator has the responsibility to ensure that only valid data is stored in the database and to provide ways to define what is valid data and what is not. The data validators are also defined in the interface for the models. Attributes of the subclass that represent columns in the table of the model can be validated on specific criteria. Each time data is created or updated the validation criteria are checked. If the validation criteria are not met, the operation will fail.

Database Migrator

The database migrator is responsible for providing ways to alter the database schema in a consistent and easy way. Migrations can be interfaced in two ways:

1. Manual definition in a ruby domain specific language.
2. Automatic generation from a generator. After a migration is defined it can be applied by calling a command in the command line.

Renderer

The renderer is responsible for rendering an HTTP response body and to give the HTTP response. The render method that is called from a controller is the interface to the various rendering techniques used. There are 8 rendering options described in the variability overview.

Rails Generators

The generators have the following responsibilities:

1. Set up a Rails application.
2. Generate boilerplate code.
3. Set up user-defined generators.
4. Perform database migrations.

The generators form an external entity that needs to be called from the command line.

Gem integrator

The gem integrator has the responsibility to integrate gems into Rails applications. Most users use bundler for the installation gems. Bundler also keeps track of the versions and dependencies of gems. Gems can after installation be added to the application by using the ‘require’ command in the Ruby code.

Debugger

The debugger is responsible for logging information and for tracking down problems and issues in the application that tests do not spot. There are three interfaces for the debugger.

1. There are methods in Rails that can be used to provide debug information.
2. Debug information can be sent to the Rails logger.
3. The Byebug gem can be used to set breakpoints in the code and analyse the code at runtime.

Asset provider

The asset provider has three responsibilities:

1. Concatenate CSS and JavaScript assets.
2. Asset minification and compression.
3. Provide assets to end-users.

Assets are placed and interfaced in the `app/assets` directory. The sprocket middleware serves the files by default in that directory.

Common Developer Scenario

To show how the system behaves we walk through the implementation of a Ruby on Rails forum application. The requirements for this application are simple. There are multiple users that log into the system and can post messages on the forum board. At first, the application is created by using a Rails generator. The application router contains paths for the login screen and for the forum board. There is a `User` model class and a `Message` model class. Validators are added to the model classes to check that the message and username fields are of limited length. When an end-user posts a message the router instantiates a new controller instance and calls an action method of that controller. In the method, a data storage operation is called. However, as the model has a validator on the message column, the message length is checked prior to storing the data. Then, the render method is called and renders an HTML response based on the forum board template that is sent to the browser. The browser requests CSS assets that are mentioned in the HTML template. These are served by the asset pipeline.

Development Viewpoint

This section presents the development viewpoint of the Rails framework, providing information of how Rails is structured, developed and released.

Module Organization

Rails uses the Model-View-Controller (MVC) architectural pattern to enforce maintainability and to correctly decouple responsibilities for each layer, namely the *model*, *view*, and *controller* layers [6]. Other modules that are not part of the MVC layers are categorized as *core utility* or *utility*. The module structure, in Figure 3, gives an overview of the layers and their interlayer dependencies.

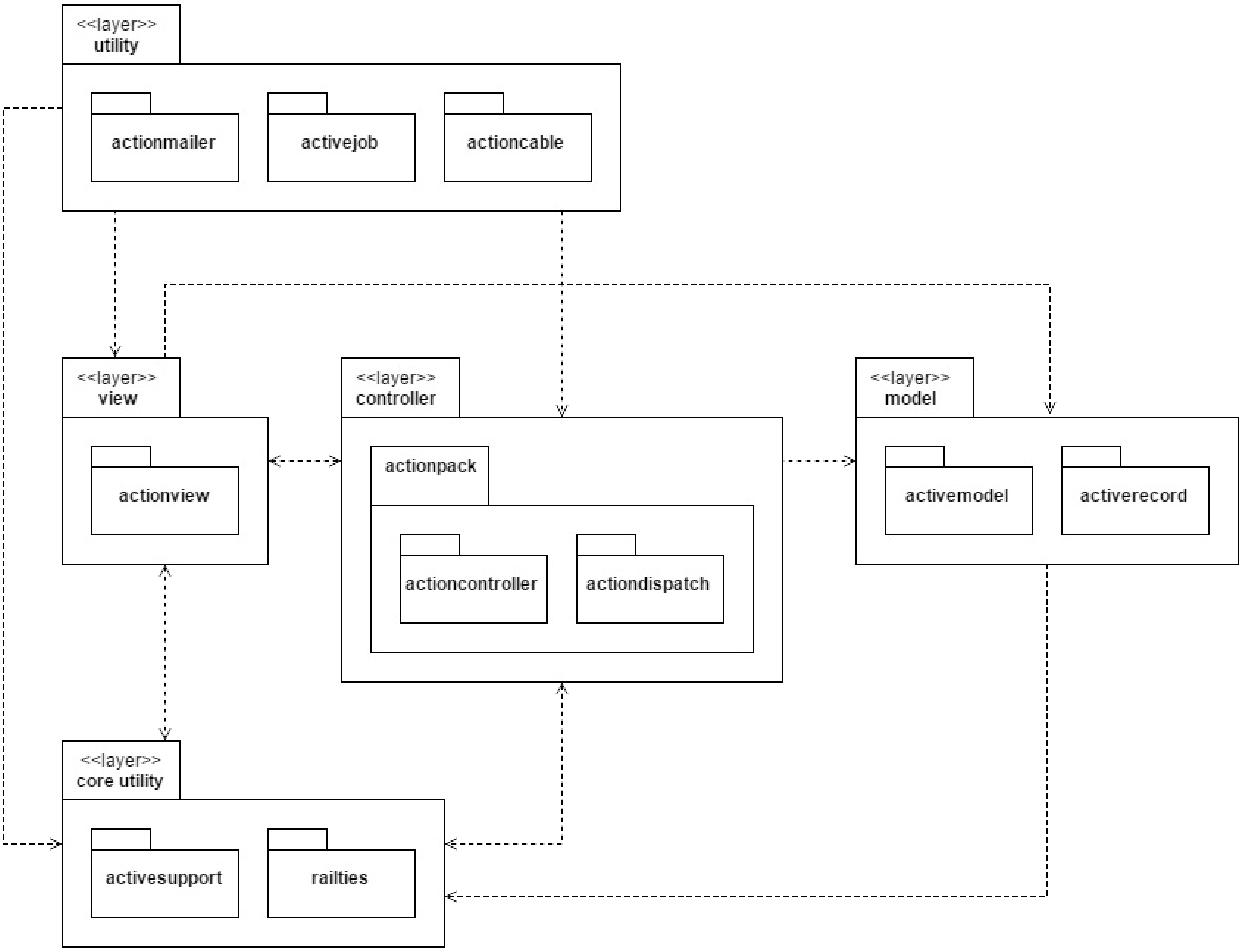


Figure 3: Module structure model of Ruby on Rails with interlayer dependencies.

1. The **model layer** consists of modules that encapsulate database and business logic.
2. The **view layer** contains all logic related to view template lookup and rendering, and provides view helpers that assist when building HTML forms, Atom feeds and more [8].
3. The **controller layer** is chiefly responsible for routing and handling web requests.
4. The **core utility layer** contains a collection of utility classes and standard library extensions which all other layers depend on. The `railties` module contains

functionality to *glue* all modules together [11].

5. The **utility layer** contains utility modules that provide optional functionality, such as sending and receiving emails, running background tasks and WebSockets integration.

Standardization of Design

In this section, we describe the common components and design of major modules, e.g. Action Pack, Action View and ActiveRecord, and the recommended development approach to get contributors started.

In the Module Organization, we have seen that every module depends on Railties and Active Support. For this reason, we clarify the role of these two modules in Rails.

> Railties is responsible for *gluing* all frameworks together. [11]

Every major module in Rails is, in fact, a Railtie. Modules define their own initialization and generators using Railtie. Furthermore, Railties is responsible for managing the Rails command-line interface [11].

> Active Support is a collection of utility classes and standard library extensions that were found useful for the Rails framework. [10]

One essential utility that Active Support provides is `ActiveSupport::Autoload`, which allows developers to lazily load constants.

Rails provides developers a standard development environment, which can be installed using Vagrant and Virtualbox and a contributing guide on coding style conventions Finally, Batsov has written an extensive style guide for Rails.

Standardization of Testing

Rails promotes test-driven development (TDD) [13]. To enforce this, Rails' Active Support module provides a test class, `ActiveSupport::TestCase`, that should be extended by every other test class. The `TestCase` class utilizes the MiniTest test framework and defines additional or enhancing functionality, such as assertions and fixtures.

Once a change in code is completed, it is recommended to run the test suite(s) belonging to the affected module(s). If all tests pass, one is allowed to make a pull request on GitHub. This will, then, trigger a continuous integration service, namely Travis CI, to ensure that the proposed changes do not introduce unexpected failures [13].

Instrumentation

Rails provides severals means for source code instrumentation; Logger, Log4r, Byebug, and more.

Rails provides the `ActiveSupport::Logger` class to log information, which can be substituted by Log4r, to be utilized for debugging purposes. Both tools provide the means to alter the level of detail logging for messages, which can improve the performance of a system.

When one is not able to detect a bug in the code using Logger or Log4r, one is suggested to use a Byebug, which allows developers to introduce breakpoints into the code.

Source Code Organization

In this section, we describe the structure of the source code, which specifies how code is structured, built, tested, released, and deployed [18].

The code structure of Rails is explained in the Module Organization. However, we have not yet described the structure of a typical module. Each module contains three directories, namely `bin` , `lib` , and `test` . The `bin` directory contains executable Ruby scripts. The `lib` directory contains source code for the module, i.e. functionality is implemented in this directory. The `test` directory contains all test code, which ensures that code adheres to the desired functionality [13].

Rails uses Rake as its build tool. It is used to define tasks that can be executed, such as testing, generating documentation, preparing releases, installing gems, and more. These tasks can be defined in a `Rakefile` . Additionally, Rails uses Bundler to manage its dependencies. Dependencies are defined in a so-called `Gemfile` , which enables one to install all dependencies using one command: `bundle install` .

Once a developer is done writing code, (s)he has to test the written code and check for regressions. This can be done locally using Rake, which defines a `test` task. Subsequently, if the tests pass locally, one can push the changes to the Rails repository hosted on GitHub to trigger the Travic CI service. Also, Rails has a rails-bot that automatically assigns the *appropriate* developer to review the changes. This way, Rails attempts to ensure that the changes do not introduce any regression.

Finally, Rails has defined a Rake task to release Rails and its modules on RubyGems, a gem hosting service. RubyGems allows Rails to define metadata, such as name and description, and its dependencies in a `*.gemspec` file. This way, Ruby developers can install a module and its dependencies effortlessly with Bundler. Further information about releasing can be found in the `RELEASING_RAILS.md` file.

Variability Viewpoint

Variability is the ability to tailor the system in different ways, according to different needs. Instead of making a software according to stakeholder requirements, a system can come in different *shapes* to suit its stakeholders.

Rails is a framework for the many, but the many do not necessarily want the same thing. Providing variability is thus an important goal for Rails, to suit its stakeholders' needs.

Features

In this section, we have chosen a set of configurable features. This list is not exhaustive but shows an accurate example of Rails' variability and modularity.

Web Server

A web server handles HTTP requests, performs calculations and returns the appropriate response to the client.

Rack provides an interface which is compatible with all kinds of web servers [20, 19, 3]:

- **Passenger** has built-in deployment error resistance and has both an open source and enterprise variant.
- **Puma** has a tiny footprint in both size and execution resources, most importantly memory, consumption and supports multiple threads.
- **Thin** claims to be the most secure, stable, fast and extensible Ruby web server.
- **Unicorn** has both a rich set of features, but delegates whatever responsibilities it can to other resources.
- **WEBrick** is the standard for a new Rails project, thus very simple to start using for development.

The choice of web servers can be found in the `Gemfile`. The decision of which web server to use is made directly after the `rails server` command is executed, thus the binding time is at *load-time*.

Request Handling

Web request and responses handling in Rails is done by the Action Dispatch module and provides a range of middleware components, from which SSL and session cookies are vital when it comes to security.

ActionDispatch::SSL contains all the logic to enable SSL with a list of configuration options, like `config.force_ssl`, the `secure` flag and HTTP Strict Transport Security (HSTS). The ActionDispatch::Cookies component sets the cookies for requests and contains three main configuration flags regarding cookie signing and salting.

For every request handled, the Rack `call` method is called by one of the Action Dispatch components, which then routes the request to the correct method and ensures that an appropriate response is sent back with cookies if so desired, thus the binding time is at *run-time*.

Object-Relational Mapping

Object-Relational Mapping (ORM) is a technique that connects rich objects to databases [14].

Because Rails is built in a modular manner, it can be used with many ORM frameworks:

- **Active Record**: Default ORM shipped with Rails and thus favors conventions over configurations.
- **Sequel**: A plugin-based system, allowing plugins to extend its functionality [5].
- **Ruby Object Mapper**: Advantageous over Active Record due to its capabilities to use data providers such as REST API, NoSQL databases, etc.
- **MongoMapper**: Maps objects to a NoSQL database, namely MongoDB. **ActiveResource**: Similar to Active Record, as it follows the Rails philosophy but is specially developed for a REST API.

Also, Rails applications can be developed without an ORM, i.e. developers can write pure database queries.

The ORM variability is achieved by the Active Model gem that provides an interface which external ORM developers should adhere to, such that their ORM can be integrated into Rails. If a user wants to use another ORM, (s)he should indicate in `Gemfile` and `config/application.rb` which ORM they want to use by including a custom ORM gem. Due to the fact that one has to configure an ORM in configuration files, the ORM is bound at *load-time*.

Database

Based on what ORM framework is used, one can choose a database system that suits one's requirements best: **SQLite**, **PostgreSQL** or **MySQL**.

Furthermore, one is able to use different databases, such as MongoDB and REST web services, by using alternative ORM gems, and can use many other databases with Rails, e.g. one can install Active Record adapters for Firebird, DB2, etc. Note that one is also able to use Rails without a database and, therefore, will only be able to store data in memory.

The user defines the used database in the `Gemfile` and `config/database.yml`, a configuration file, which is read once when a Rails application is loaded, resulting in binding at *load-time*.

Rendering

Rendering is the process of turning a template or string into an HTTP response with the appropriate format rendering, being one of the following: nothing, ERB template, Builder template, text, HTML, JSON, XML or Javascript.

The rendering is started by calling the `ActionController::Base#render` method. The user decides the controller logic and thus when a render method is called, so bound at *run-time*.

Caching

There are three types of caching in Rails for templates: page caching (always done on disk), action caching and fragment caching. According to the Rails Guides [15], there are different options available for the storage of action and fragment cached data: Memory store, File store, MemCache store, Ehcache store, Null store and Custom store.

The methods in the cache store class are called inside Rails to interact with the cache store, which is configured at the start of the application and thus is bound at *load-time*.

Asset Pipeline

The asset pipeline provides tools and mechanisms by which JavaScript files, stylesheets, and images can be processed and prepared for use by the browser [4].

`Sprocket-rails` is responsible for plugging the asset pipeline inside a Rails application. When enabling the asset pipeline with the `config.assets.enabled` flag, the `uglifier`, `sass-rails` and `coffee-rails` gems are installed alongside. Sprockets concatenates all JavaScript files into one `.js` file and all CSS files into one `.css` file, and additionally provides tools for compression. When it comes to customising the pipeline itself, we can look at significant features that can be configured. CSS and JS compression can be done by setting `config.assets.css_compressor` and `config.assets.js_compressor` to match the compressor gem.

Developers may choose different languages in order to code assets. Supported languages include Sass for CSS, CoffeeScript for JavaScript, and ERB for both by default [4], which all have to be preprocessed.

The languages supported by Rails are precompiled during preprocessing time, before the loading of the application. However, the binding precompilation and all configurations relating to ERB files or compressions tools are recorded and loaded when the server loads, thus the binding time is at *load-time*.

Feature Dependency Analysis

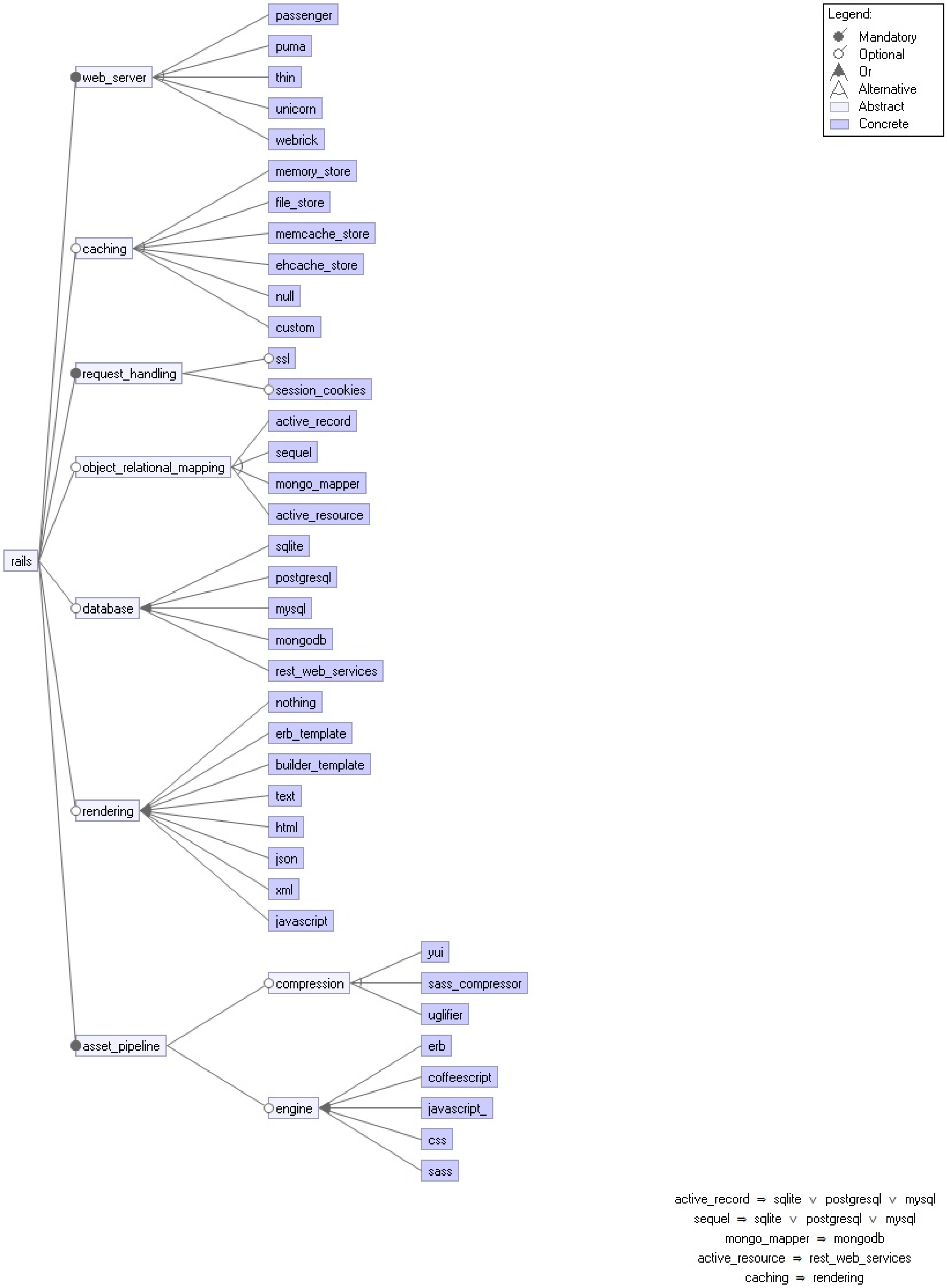


Figure 4: Feature model of Rails, created in FeatureIDE.

Rails is built in a modular way, allowing the user to interchange most components in the system, thus, reduce the number of dependencies between features. As can be seen in Figure 4 There is a dependency between Object-Relational Mapping frameworks and database systems. ORM frameworks are developed in such a way that they only support particular databases. For example, using MongoDB is facilitated by the Mongo Mapper framework. There is a dependency between page caching and rendering. The pages or fragment of pages first need to be rendered before they can be cached. Moreover, when looking at the asset pipeline, we observe that it **used to be** a direct dependency to Action Pack. However, it is now a separate module that can be added to Rails depending on the developer's wishes.

Security Perspective

When developing for the web, one of the first concepts that should come to mind is security. Many frameworks, including Rails, provide useful helper methods that ensure certain security aspects, but security ultimately depends on the people using the frameworks [17].

Security is defined by Rozanski and Woods [18] as the set of processes and technologies that allow owners of resources in a system to reliably control who can read, change or execute which resources. Resources are at the center of the system's security, and *security mechanisms* enforce the *policies* that define how, and by which *principals*, these resources can be accessed.

Security Concerns

Taking a look at the security perspective defined by Rozanski and Woods [18], we identify how the system under analysis, namely Rails, deals with the most important concerns this perspective addresses. The most important concerns include:

- Managing and securing of **resources**.
- Access rights of entities, also known as **principals**.
- Designing of security **policies**.
- Security **threats** to the system.
- The CIA triad of **confidentiality**, **integrity**, **availability**, and **accountability**.
- Threat **detection and recovery** from attacks.
- **Security mechanisms**.

Security Concerns and Rails

Rails is a framework that facilitates building web applications. We can, therefore, argue that it is not the responsibility of Rails to ensure the security of resources, *authentication* and *authorization* of principals, and the design of security policies. However, Rails does provide its users, who are developers of Rails applications, with tools and guidelines on how to handle these aforementioned concerns in their applications [17]. Policies can easily be designed with the help of Rails sessions, provided user management tools and admin security.

Regarding threats that may arise in Rails applications, Rails itself keeps in mind details about the possible attacks. The guides provide with helper methods and guidelines on how to protect Rails applications against e.g. cross-site request forgery attacks (CSRF), cross-site scripting attacks (XSS) and session hijacking. Since Rails was designed to help developing database-backed web application, SQL injections are also a potential threat. It also gives a clear definition of what these threats are, how the possible attacks are performed, and how to identify them.

Rails attempts to make it possible for users to ensure the CIA triad, with the accountability addition, by means of sessions to ensure confidentiality and logging to ensure integrity. Additionally, with the use of sessions, an action can be traced back to the principal who performed it, therefore ensuring accountability. Availability is, however, another story, as it is the user's responsibility to make sure the application is available for end-users, Rails has no say in this. When end-users authenticate on Rails applications, they will have their personal session assigned to them, with information only divulged to the specific session token. Rails provides its users with logging tools that log activities inside the application, which is useful to check whether data was *inadvertently* changed.

Rails' logging system is a good tool for detecting attacks on integrity and accountability, as it makes it possible for Rails developers to track the resource flow on their system. However, recovering from such attacks is not really part of Rails. There are no security monitoring mechanisms incorporated in Rails.

Conclusions and Discussions

After this deep analysis of the Rails project, it is worth noting the following three points:

- Rails is a large center of interest, which is always evolving to provide better or newer features for web development.
- Rails has a large and friendly community, who is always eager to welcome new contributors.
- Convention over Configuration is what makes Rails a suitable framework to start with.

The lesson to learn from this is that Rails is easy. This is why so many people and companies use it; this is why GitHub uses it! It reduces advanced configurations to one set of flags, it provides immediate database support, it makes it possible for beginner developers and startups to make advanced web applications that kick off their business plan.

With that being said, there is one thing we should consider: Rails is a project from over ten years ago. Therefore, it is reasonable to say that many newer frameworks have followed in the footsteps of Rails, and have made their approach even better. Hence, we have to keep in mind that Rails is getting old, and more recent advancements in web development are slowly replacing it.

Moreover, Rails has one little aspect that is possibly problematic: the Ruby language. Rails lacks performance mainly because of Ruby. As easy to understand as it may be, Ruby remains an interpreted language, which, performance-wise, is not efficient.

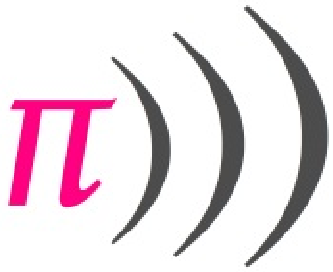
Are you new to the open source community and interested in contributing to Rails? Take a look at Rails and the guidelines to contribute to the Rails project. The Rails team will welcome you with open arms, and will provide extensive feedback on even the smallest contributions you make.

References

1. BuiltWith. Ruby on Rails Usage Statistics. URL: http://trends.builtwith.com/framework/Ruby-on-Rails, 2016.
2. Chande, G. What is ‘Rack’ in Ruby/Rails? URL: http://blog.gauravchande.com/what-is-rack-in-ruby-rails, 2013.
3. Engine Yard. Rails Server Throwdown: Passenger, Unicorn or Puma? URL: https://www.engineyard.com/articles/rails-server, 2016.
4. Launch School. Everything You Should Know About the Rails Asset Pipeline. URL: https://launchschool.com/blog/rails-asset-pipeline-best-practices, 2014.
5. Marohnić, J. Ode to Sequel. URL: https://twin.github.io/ode-to-sequel/, 2015.
6. Mejia, A. Ruby on Rails Architectural Design. URL: http://adrianmejia.com/blog/2011/08/11/ruby-on-rails-architectural-design/, 2011
7. Neukirchen, C. Rack: a Ruby Webserver Interface. URL: https://rack.github.io/, 2012.
8. Rails. Action View. URL: https://github.com/rails/rails/blob/master/actionview/README.rdoc, 2015.
9. Rails. Active Model – model interfaces for Rails. URL: https://github.com/rails/rails/blob/master/activemodel/README.rdoc, 2015.
10. Rails. Active Support – Utility classes and Ruby extensions from Rails. URL: https://github.com/rails/rails/tree/master/activesupport, 2016.
11. Rails. Railties – Gluing the Engine to the Rails. URL: https://github.com/rails/rails/blob/master/railties/README.rdoc, 2014.
12. Rails. Ruby on Rails. URL: http://rubyonrails.org/, 2016.
13. Rails Guides. A Guide to Testing Rails Applications. URL: http://guides.rubyonrails.org/testing.html, 2016.

14. Rails Guides. Active Record Basics. URL: http://guides.rubyonrails.org/active_record_basics.html, 2016.
15. Rails Guides. Caching with Rails: An overview. URL: http://guides.rubyonrails.org/caching_with_rails.html, 2016.
16. Rails Guides. Ruby on Rails Guides. URL: http://guides.rubyonrails.org/, 2016.
17. Rails Guides. Ruby on Rails Security Guide. URL: http://guides.rubyonrails.org/security.html, 2012.
18. Rozanski, N. Woods, E. Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives. Addison-Wesley, 2012.
19. Tezer, O. S. A Comparison of (Rack) Web Servers for Ruby Web Applications. URL: https://www.digitalocean.com/community/tutorials/a-comparison-of-rack-web-servers-for-ruby-web-applications, 2013.
20. The Ruby Toolbox. Web servers. URL: https://www.ruby-toolbox.com/categories/web_servers. 2016.

Sonic Pi - The Live Coding Synth for Everyone



An in-depth analysis by Tom den Braber, Jeroen Castelein, Michel Kraaijeveld and Paul de Goffau, February-March 2016.

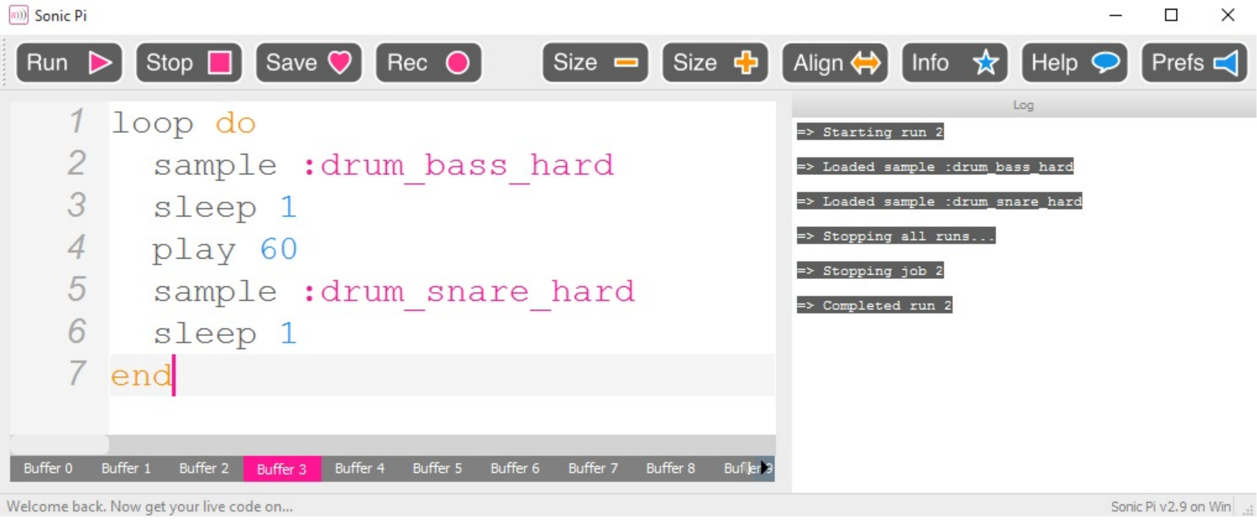
Abstract

In this chapter we provide an analysis of Sonic Pi. Sonic Pi is an application in which code can be written to create music. Through examining the source code, the existing documentation and by contributing to the system, we gained insights which we used to describe a set of relevant views and perspectives. These views and perspectives are based on software architecture standards. They range from stakeholder analysis, to user functionalities, to the inner development view. We round off with a small section on our experiences with contributing to the project.

Table of Contents

Introduction

Sonic Pi is a free offline application where users can write their own code to produce music. It is designed to provide an accessible programming platform for children or others with little programming experience. Rather than having to create your first Java class with `public static void main(String[] args)` (what?) to get close to printing your first `Hello World` to the screen, all you have to do is write down `play 60` to hear your first beep. Not impressed yet? Add a loop, drum samples and some time between the samples. That's all you need to get your first never-ending beat rolling. The code is entered within the Sonic Pi application. A screenshot of the GUI of the application can be found in Figure 1.



```
loop do
  sample :drum_bass_hard
  sleep 1
  play 60
  sample :drum_snare_hard
  sleep 1
end
```

```
=> Starting run 2
=> Loaded sample :drum_bass_hard
=> Loaded sample :drum_snare_hard
=> Stopping all runs...
=> Stopping job 2
=> Completed run 2
```

Welcome back. Now get your live code on...

Sonic Pi v2.9 on Win

Figure 1: Screenshot of the Sonic Pi application, with code for the never-ending beat

Educational tools helping users learn how to code are provided in the form of tutorials. Experienced users have been able to write existing or new songs, which they may share through code or sound. Another feature that Sonic Pi is proud of, is live coding. By creating a `live_loop`, users can modify their code on the go without having to stop the music in between. This allows Sonic Pi to be used for live performances, and a small group of artists use Sonic Pi to perform at clubs and parties. The development of Sonic Pi is done on GitHub and its code is completely open source. Here, the developers come together to discuss issues, new features and changes to the system. In the rest of this chapter, we present our analysis of Sonic Pi. We offer several architectural views and perspectives, inspired by the book on Software Architecture by Rozanski and Woods [1]. After reading this chapter, you will have a core understanding of the functionalities of Sonic Pi, its context, the stakeholders involved and the design choices made. On top of that, you will be able to use this knowledge to start contributing to the project yourself.

Functional view

Before we introduce all of the functionalities Sonic Pi has, it is good to mention what Sonic Pi aims to be. This can be learnt from the 'motto' as stated on the official website [2]:

> Learn to code creatively by composing or performing music in an incredible range of styles from classical to algorave.

Here, we already have a large portion of that what Sonic Pi wants to be summed up. The primary goal is education: Sonic Pi offers a method for learning to code. However, Sonic Pi also strives to be a musical instrument, and not just any musical instrument: it should be possible to make any kind of music you want to make, no matter how complex. This can also be seen in the repository of the project [3]:

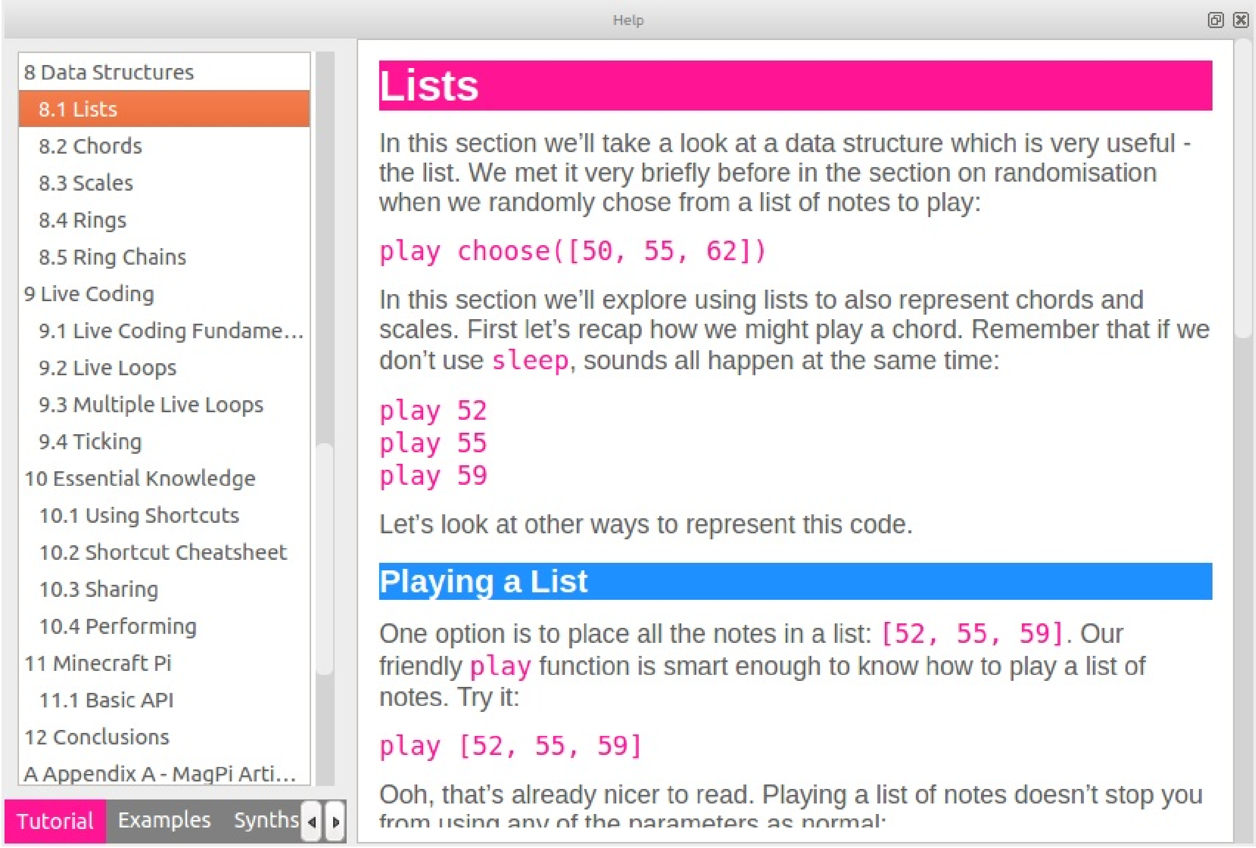
> In addition to being an engaging education resource it has evolved into an extremely powerful and performance-ready live coding instrument suitable for professional artists and DJs.

Functional Capabilities

Now that we know what the goals of Sonic Pi are, we can see which functionalities are offered to reach these goals. Because some functionalities are strongly related to each other, we categorize them into four categories: education, code, music and accessibility.

Education

Sonic Pi provides lessons within the application. Users can click on the 'Help' button in the GUI and the tutorials and examples are shown at the bottom of the screen. The users can follow these lessons to learn programming and learn about programming concepts. The lessons are written in such a way that they can be followed by aspiring programmers, even if those aspiring programmers are still children. An example of such a lesson can be seen in Figure 2. In this specific example, the student can learn about the list data structure and how it can be used in Sonic Pi.



Lists

In this section we'll take a look at a data structure which is very useful - the list. We met it very briefly before in the section on randomisation when we randomly chose from a list of notes to play:

```
play choose([50, 55, 62])
```

In this section we'll explore using lists to also represent chords and scales. First let's recap how we might play a chord. Remember that if we don't use `sleep`, sounds all happen at the same time:

```
play 52
play 55
play 59
```

Let's look at other ways to represent this code.

Playing a List

One option is to place all the notes in a list: `[52, 55, 59]`. Our friendly `play` function is smart enough to know how to play a list of notes. Try it:

```
play [52, 55, 59]
```

Ooh, that's already nicer to read. Playing a list of notes doesn't stop you from using any of the parameters as normal:

Figure 2: An example of a lesson that can be found within Sonic Pi

Code

Sonic Pi provides its own Domain Specific Language (DSL). This language can be used to 'describe' sounds and subsequently play them. The Sonic Pi DSL is a collection of functions written in the Ruby programming language. Sonic Pi can also transform the code that is written into sounds. In order to be able to run this code, Sonic Pi has a code editor in which code from the DSL can be inserted. Within the GUI, it provides some standard programming tools: users will see the exceptions their code throws, they can inspect the log-output and code can be automatically aligned. The code editor is very basic; for example, it does not contain a debugger.

Music

Although being able to write code in Sonic Pi is nice, it also has to be translated into music. This functionality is also present, making Sonic Pi a musical instrument. It is done by providing a number of synthesizers and some functions for timing and concurrency. Sonic Pi itself does not actually make music: it uses an external platform called SuperCollider [4]. This does not mean that Sonic Pi does not do anything concerning the playing of music. Sonic Pi instructs SuperCollider on how and what it should play in terms of timing, beats per minute and what instruments to play among others. To be able to use Sonic Pi for 'live coding', Sonic Pi is able to keep playing music while the code changes, and smoothly bring the changes into the currently playing musical composition.

Accessibility

The accessibility aspects of the application are strongly related to the goal concerning education. To be able to be truly educative, it is also important to be easily accessible so that people will actually use the software. Therefore, Sonic Pi provides a user interface which is internationalized to a great extent. Almost all of the text shown in the GUI is loaded at runtime, translated to the default language of the computer running Sonic Pi if available. Another aspect of accessibility is that the application comes with an installer, which is very easy to use.

As the name of Sonic Pi indicates, one of the core goals is to be able to run smoothly on a Raspberry Pi. The Raspberry Pi is a very cheap computer, so being able to run on the Raspberry Pi enlarges the possible userspace of Sonic Pi. Sonic Pi can however be run on all major platforms to make it possible for everyone with access to a computer to learn programming.

Internal Structure

Now that it is clear which functionalities Sonic Pi offers, we can have a first look at how Sonic Pi implements these functionalities. A high-level overview can be seen in Figure 3.

There are two non-obvious choices that deserve some explanation: the choice for using SuperCollider as synthesizer tool and the choice for the overall architecture of having a client and a server. The choice for SuperCollider can be explained by looking at another project where Sam Aaron, the founder of Sonic Pi, was involved. In 2010, a project called Overtone [5] was founded, which also heavily relies on SuperCollider. Sam Aaron is also one of the main contributors of Overtone and has contributed to this project from the start and thus already had experience with SuperCollider when starting Sonic Pi in 2013. It is likely that this experience lead to the choice for using SuperCollider in Sonic Pi. The second choice can possibly be explained by looking into the future: one of the ideas for contributions that can be found in the Sonic Pi repository is to synchronize multiple instances of Sonic Pi over the internet [6], so that multiple artists can form an 'orchestra'. The client/server architecture could be a first step towards that goal, albeit that currently every client has an own dedicated server.

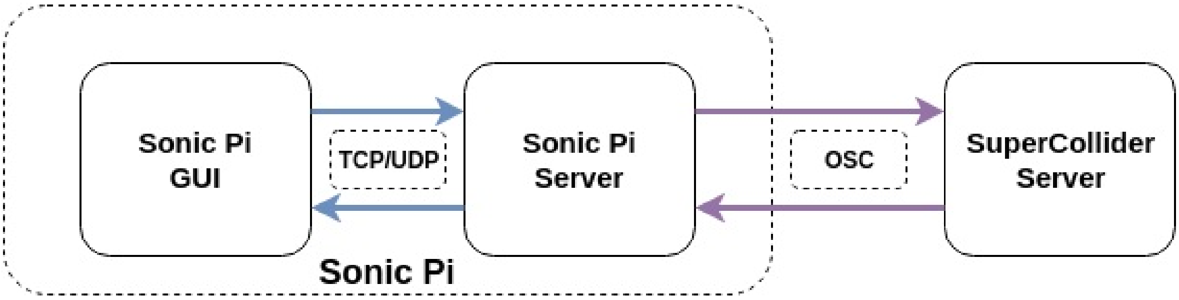


Figure 3: Internal structure of Sonic Pi

The GUI takes care of all interaction with the user. The Sonic Pi server provides all the business logic: the parsing of the DSL and sending instructions to the SuperCollider server.

The arrows in the diagram show the message flow. The GUI communicates with the Sonic Pi server over TCP or UDP. The Sonic Pi server communicates with the SuperCollider server using the OSC protocol. This protocol can be used for networking sound synthesizers, computers and other multimedia devices [7].

Stakeholders

In order to understand why certain design choices have been made, and what roles people have taken in this project, we will describe the stakeholders that have been involved in the project. These are separated into five groups: developers, suppliers, communicators, users and investors.

Developers

The most important group of stakeholders is the developers. They are the ones who actively work on the project and use the application to come up with improvements and new ideas. The team of developers is quite small. Sam Aaron is by far the main contributor and he responds to all issues and merges most of the pull requests. Joseph Wilk (@josephwilk), Xavier Riley (@xavriley), Jeremy Weatherford (@jweather) and Hanno Zulla (@hzulla) are other important developers that often join the development discussions or create issues/pull requests. They are mostly active in collaboratively thinking of a solution, after which it is implemented. All of them have a great interest in Sonic Pi, whether as an educational tool or a music making tool which motivates them to develop further.

Suppliers

Sonic Pi is designed to run on the Raspberry Pi platform. The low price of the Raspberry Pi allows users to invest little money, while getting all of the features that Sonic Pi provides. Its small size also allows users to carry it along easily. This is what makes the Raspberry Pi foundation an important supplier. Raspberry Pi has also included the Sonic Pi application in their Jessie distribution [8].

The SuperCollider project is the supplier of the sound for Sonic Pi, as it turns instructions into sound.

Communicators

This group consists of the Raspberry Pi Foundation, translators and guide creators. The Raspberry Pi Foundation has supported its development by offering learning material on their website [9]. A lot of people are involved by translating the tutorial that is embedded in the application. Developers help each other by creating documentation. The most useful documentation consists of guides to build the system on different operation systems.

Users

There are two groups of users, those who use it for education and those who use it for live performance.

- Education. Teachers use Sonic Pi to learn their students how to code interactively. It is not clear how many teachers do this, however the Raspberry Pi lessons have been trialled at a number of schools as part of the Computing curriculum in the UK. In this blog by one such teacher, he portrays how well Sonic Pi is received by his students.
- (Live) musical performance. With Sonic Pi and live coding, Sam Aaron wants to introduce the computer as an instrument. He frequently performs at events and on livestreams where he codes music. This is well received by the audience as it is quite impressive. Other artists have put some music up on for instance SoundCloud. It is interesting to see that also source code is uploaded. An example is the start of the famous song "Let it go" from Disney's Frozen.

Both of these groups use the software for learning and enjoyment. They may encounter bugs or have ideas for new features, at which point they can approach the developers by creating an issue on GitHub.

Investors

The Sonic Pi project originates from a project funded by the Broadcom Foundation, executed by Sam Aaron at the University of Cambridge [10]. The aim of this project was to create a product that would help teachers use the Raspberry Pi to teach computer science.

As time went by, the interest for Sonic Pi grew, now funded by the Raspberry Pi Foundation. They decided to improve Sonic Pi and design a v2.0. This project was called *Sonic Pi: Live and Coding*, and was partnered by Cambridge Junction and the Raspberry Pi Foundation. From Nesta, a charity located in the UK dedicated to support ideas that can help improve our lives, they received a large donation to realize this project.

Classification of stakeholders: Power vs. Interest

All these stakeholders and their interests fit into a power-interest diagram, which can be seen in Figure 4. Power indicates that a stakeholder has a lot of influence on the project. Interest indicates that a stakeholder has a high interest in the prosperity of the project. It is interesting to see how the stakeholders of any project actually interact with the development, should they be satisfied by developers or should the developers aim to accommodate to powerful stakeholders.

The stakeholders are placed according to their position within the project:

- **The developers** have high interest and high power, as they manage the whole project with passion. Their ideas and effort allow Sonic Pi to grow.
- Sam Aaron is on payroll at **Cambridge University** where he is a research associate. The Sonic Pi project started there, but after that they were never too involved with the development process.
- **SuperCollider** is an application that is used by, but does not rely on, Sonic Pi. It has little influence on Sonic Pi's growth.
- Stakeholders with low power and high interest are the **investors**, **suppliers** and **translators**. They want to use the application itself, or want to see it flourish because they think it might be interesting for the society.

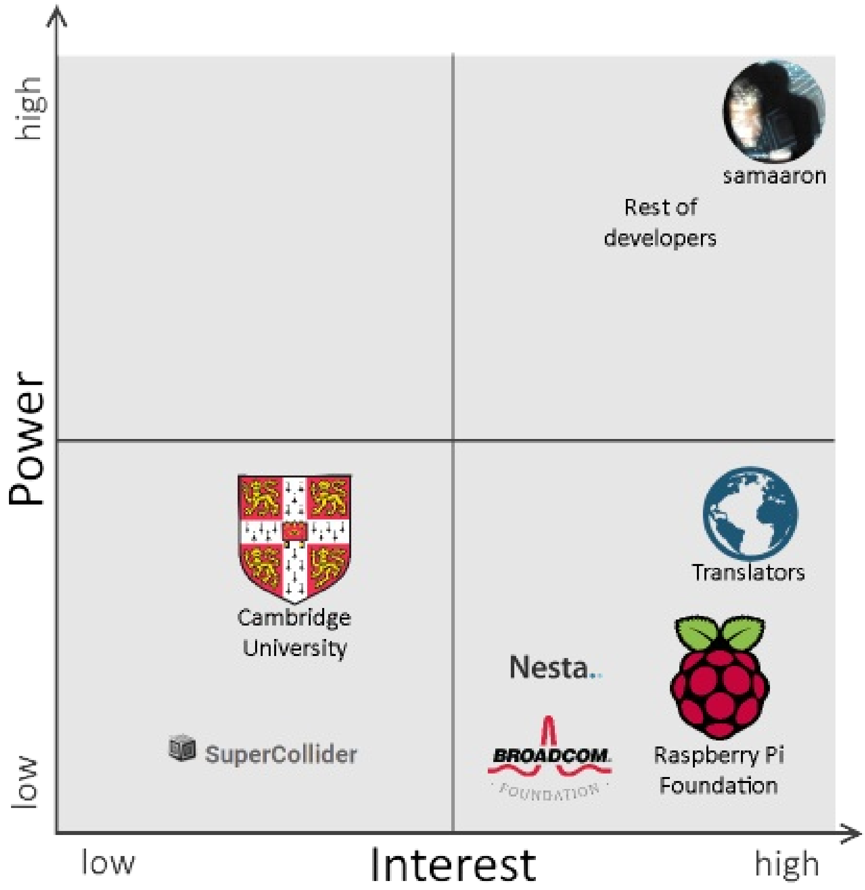


Figure 4: Power interest diagram of the Sonic Pi stakeholders

Context view

In Figure 5 Sonic Pi is depicted in relation to its context. You can see the two groups of users, students and music performers, depicted at the right of the diagram. The users enter code into the GUI. This code is sent to, and interpreted by a server that runs a Ruby interpreter. The server calls the SuperCollider application to play sounds. The SuperCollider application processes these calls, and sends the required sound bits to the local sound system. This sound is heard by the user. The required sounds have mostly been created by an application called Overtone, where performers can create new music samples. The Sonic Pi system is maintained and improved by the developers. These developers are supported by documentation made by the communicators. The developers are funded by some companies, depicted by the logos of Nesta and the Broadcom Foundation. The developers base their maintaining and improvements on the issues, mostly created by teachers that interact with their students, but also directly by the students. Recall that the group of users also contains performers that use the application to create music.

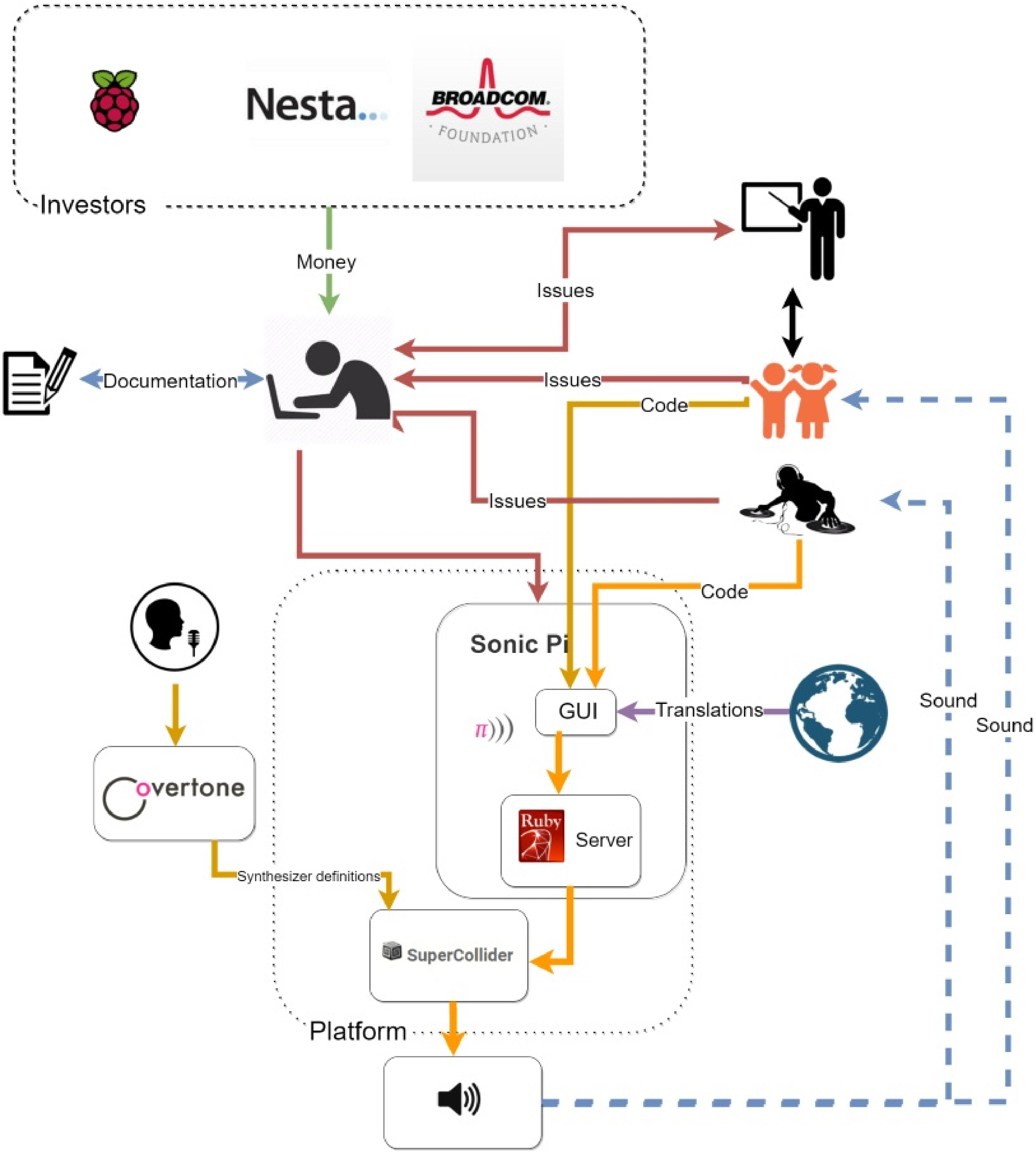


Figure 5: Context view

Development view

The development view describes the structure of the Sonic Pi system from the perspective of developers. This includes the different modules of the system, the testing and release processes and an analysis on technical debt.

Module structure model

The Sonic Pi system consists of two parts: a GUI and a server. In this section, the organization of Sonic Pi's source code is shown by giving an analysis of the modules and their interdependencies. In addition, to get a clear overview of the interactions, the modules are presented in layers.

GUI

We start by analyzing the GUI, of which the main task is to make interaction with the actual application possible via graphical elements; it should enhance the user experience. This can also be seen in the various modules that were identified. The following modules are contained in the GUI.

- **MainWindow**: The main code source, creates the GUI that is visible and controls most of the user interaction.
- **SonicPiOSCServer**: Handles the starting and stopping of the listening to the server (the other part of the system).
- **OscHandler**: Handles all the messages received, executes the relevant functions.
- **SonicPiAPIs**: Manages auto-completion.
- **SonicPiLexer**: Colors keywords based on the active theme.
- **SonicPiLog**: Adds messages to the log output in the GUI, colored based on the active theme.
- **SonicPiScintilla**: Handles all the text coloring, text inserting, layout, etc. on the main coding screen.
- **SonicPiTheme**: Implements the themes.
- **Help & doc**: Automatically generated files for the tutorials and documentation within the GUI.
- **Qt**: The external Qt framework that the GUI is built upon.
- **Qscintilla**: A port to Qt of the Scintilla[11] C++ editor control.

In addition, you can find the dependencies of the different modules depicted in Figure 6. Note that some modules depend on libraries more than one layer down, due to the use of Qt in all modules.

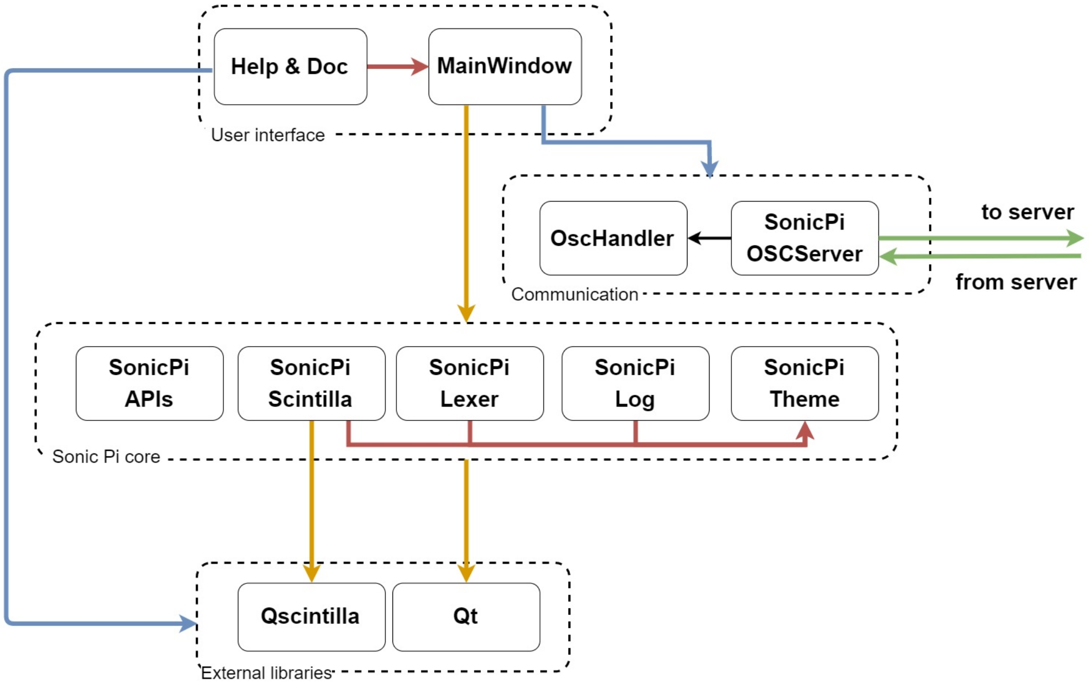


Figure 6: GUI module structure model

Server

The server part of the system contains the *business logic* of the program. The modules in the server are a collection of functional modules to execute commands from the GUI, definitions for the code to be interpreted, and the communication with the SuperCollider server.

- **sonic-pi-server.rb**: Instantiates all necessary threads to run the system on this end.
- **Runtime**: Takes transformed input from the GUI and processes this so that the necessary functions are executed.
- **Studio**: Controls the music settings, the recording, and the communication with SuperCollider.
- **OSC::UDPClient**: Listens to the GUI.
- **OSC::UDPServer**: Performs the communication with the GUI and SuperCollider.
- **OSC**: Contains the OSC encoding and decoding.
- **Core**: Has basic helper functions, some threading functions. This forms a backend to all the code.
- **Settings**: Handles user settings; loading, changing and saving.
- **SynthInfo**: Contains the information on the different synth options.
- **LangDefs**: Contains special language definitions, functions etc.
- **SoundDefs**: Contains special language definitions, functions etc. These functions also contain the execution of synths.
- **Ruby Standard Library**: The standard modules and classes that come with Ruby.
- **Ruby Gems**: Are packets of external Ruby code that are easily installable. The list of

used gems can be found in Appendix A.

You can find the dependencies of the server in Figure 7.

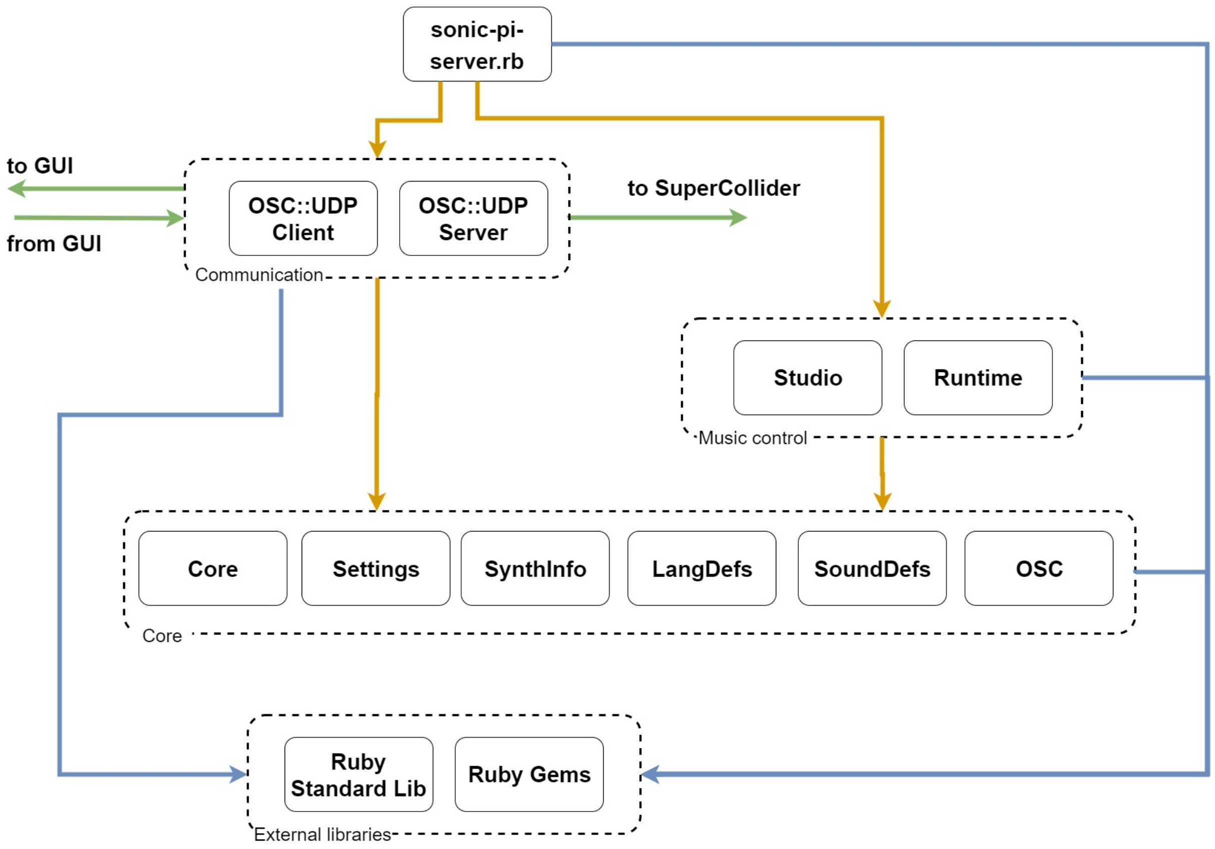


Figure 7: Server module structure model

Testing Approach

Testing is done both locally and with a continuous integration server on GitHub. At the time of writing, 109 automatic tests have been added to the system, resulting in a line coverage of 6%. This amount is not astonishing, considering there are many more classes and methods that keep Sonic Pi running. A thing to note though; the Sonic Pi source code also contains a lot of inline documentation, which are counted by the program. Therefore, the actual coverage is higher, but could not be measured with the tools available. The lack of tests can also be seen in the contribution guide, as it is not stated that tests need to be provided when creating a pull request.

Release Process

All changes are merged directly to the master branch of the project, so there is no branch available that only contains stable releases. This means that between versions, you will need to use the release tags or download an executable from the Sonic Pi website to get a stable version. Furthermore, there are no formal requirements that a release should fulfill, such as test coverage or documentation. Therefore, technical integrity is not guaranteed and releases are made when Sam Aaron decides that enough additions have been made. When a new public version is released, the current state of the master branch is tagged accordingly and new installers will be created so they can become available as downloads on the website.

Technical Debt

The extra time needed to make changes, especially to parts which were quickly and not optimally implemented, is considered as technical debt. When a project suffers from technical debt, it is more difficult and costly to maintain it properly. Several elements of technical debt can be distinguished:

- **Low test coverage**; as mentioned earlier, the system is not optimally tested as only a small part is covered. Although 100% coverage does not mean that every possible bug or defect is found, increasing the current percentage would still be a good approach.
- **Deficient documentation**; none of the files had a description of what their usage is which makes it hard to find out the exact use of some of the files. No module or method documentation is presented, which makes it difficult to understand the intended working of classes and methods. Also the source code lacks consistent commenting.
- **Addressing of issues**; multiple issues are available in the issue tracker on GitHub. Most of them are quickly addressed, but it isn't always clear why some are still unresolved. Therefore it is not clear whether the project has technical debt based on the issue handling alone.
- **Code smells**; some examples of code smells are large classes, duplicate code or a high cyclomatic complexity. Since most of the logic of the Sonic Pi program is written in Ruby, Reek [12] was used and gave a lot of valuable insights in the amount of code smells in the code. The result of this can be found in Appendix B. In addition to the code smells, there are also libraries used in the Sonic Pi project, which are not always up to date. Since around 40% of the used libraries are not up-to-date, and one is even deprecated, we can conclude that technical debt is affecting the Sonic Pi project.

Variability perspective

Practically every software product has some form of configurability in them. Whether it is obtained through different versions of the software, e.g. enterprise edition vs. free edition, or possibly by changing options within the application, such as displaying line numbers in a text editor. Sonic Pi is no stranger to this concept. Within the user interface there are many adaptable features concerning for instance logging and user friendliness. There are also different builds for different operating systems, which alter some of Sonic Pi's functionalities.

To show the configurability of the system, we display all the *variable features* offered. In the feature model (Figure 8) all of these features are displayed. Distinction is made between optional and mandatory features, as portrayed in the legend. The lighter colored boxes are abstract features containing a set of other features. The darker colored boxes are features the user may change. The model is constructed using the eclipse plugin FeatureIDE [13].

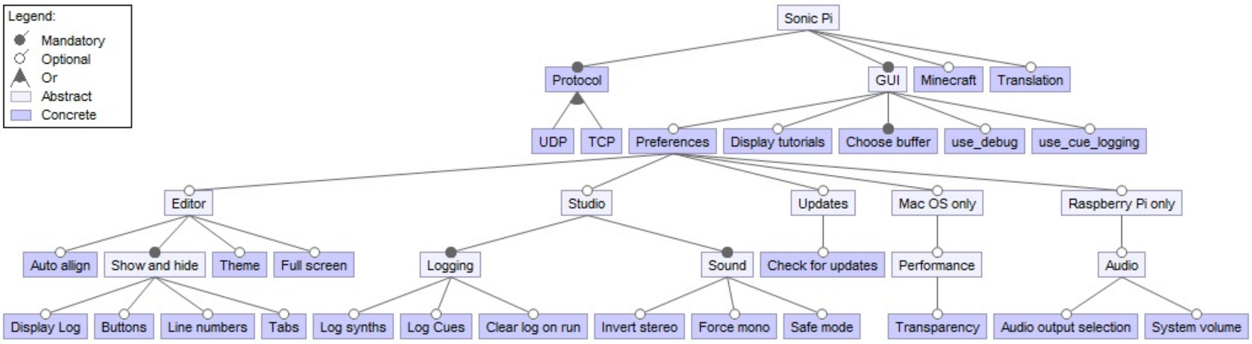


Figure 8: Feature Model, created with FeatureIDE

These variable features can be changed at different times. We distinguish compile-, boot- and runtime.

Compile-time

In the source code, directives are used to include and exclude certain lines of code when compiling for a certain operating system. A couple of features are only compiled for certain operating systems. They can be found in Figure 8 under the *Raspberry Pi only* and *Mac OS only* nodes. Since no specific features are available only on *Windows*, there is no node containing this operating system.

Boot-time

Boot-time is a special kind of runtime. In our definition, features are bound at boot-time when the feature is set when starting the program and it cannot be changed afterwards. Two features fulfill these criteria. First, the translation that is used in the GUI. When the program is booted, the system locale is used to try and find an appropriate translation. After the program has decided a language, there is no way for the user to change it. The other feature is the communication protocol (TCP or UDP) used. The server can be configured using a parameter. However, it is not possible to change this decision after the server has been started.

Runtime

All other features are decided at runtime. Whilst using the program the user can change these features. For instance, the user can switch the theme between a light and a dark theme, or they can enable/disable logging on their code runs.

Internationalization perspective

With Sonic Pi's goal to be used by children globally, internationalization (i18n for short) is an important perspective that needs to be taken into account. Although a majority of the children nowadays learns English at school early on or through interaction with the internet, most will not feel comfortable in playing with an application if it is not in their mother tongue. Sonic Pi aims to provide this comfort by translating the interface and most importantly the tutorials. At the time of writing, 9 different languages are available, from English to French to Japanese. To increase this number, the developers approach their users to contribute with a translation of their language. This is implemented by displaying a message to users from countries with no available translation with a link to the Sonic Pi repository. In the repository an extensive explanation is provided for users unfamiliar with GitHub on how to add their language.

One thing that is not translated is the code the user writes. The code can be written in unicode, and thus supports every alphabet. However, the language definition used in Sonic Pi is always in English. This is not optimal, as someone who understands what `play_pattern_timed` means through language can use it more easily than someone who doesn't.

Similar to the language definition, the Ruby runtime executing the code written by the user can produce syntax errors or other errors. These errors are shown in the GUI, but are not translated. This makes debugging difficult if the user cannot read English. A small compensation is offered by highlighting the line in the code that generated the error.

Other important elements that could still be translated are the buttons in the top bar. These buttons are based on images, containing short English terms, of which an example can be found in Figure 9. To compensate for the English labels on the images, the developers also included a small image that intuitively shows the function of the button.

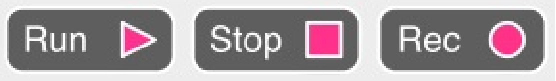


Figure 9: Some of the buttons in the GUI

Other improvements that could still be made to improve internationalization are:

- options to change the language at runtime, rather than being decided on boot.
- offering text-orientation; some languages are written right to left, rather than left to right.

How to contribute

Sonic Pi is open to contributions, but it might not be very clear on how you can contribute to it. Therefore, we devote a section to our experiences with contributing to Sonic Pi. To start off, there are no specific rules you need to comply with when writing code for the project. As a result, most of the pull requests with code changed are merged if they look good and contributing can be considered easy that way. In addition, Sam Aaron is quick to reply on pull requests, which ensures that you know what you are up to. A drawback of the lack of rules is that you don't know beforehand whether your changes fit in with the project and you need to await the initial feedback.

Due to a lack of planning, although this is improved for the upcoming version 3.0, it is not always easy to find parts of Sonic Pi you can work on. One of the things you can do, is go through the issue list and see whether you can fix any of the problems that are indicated there. You can also create an issue yourself in which you explain a feature that you want to have implemented and work on it yourself. There is a contribution guide available that lists some ideas for future Sonic Pi versions. However, it is not updated often and some of the listed features are already implemented in the meantime; making it not the best reference. The same applies to the milestones mentioned earlier, as these are not actively used.

Overall, Sonic Pi is a project to which you can contribute easily; provided that you first have found something to contribute. In the future this might be more easily, if they stick to using milestones and tagging issues appropriately.

Conclusion

Sonic Pi is an innovative application to learn coding through music. It offers tutorials and examples to get everyone started, and is even used in lessons by the Raspberry Pi foundation. Aside from teaching, it can also be used for performing. Sonic Pi has the ability for 'live coding', making it possible to compose pieces of music and play, and even edit it, live. Overall, Sonic Pi is a fun program to use and definitely a nice way to learn people how to code.

We do think that there are a couple of improvements that could be made to the Sonic Pi project. To start, the way in which pull requests are merged can be changed. Currently changes are merged when they look good, but there is no proper testing required, resulting in some merges breaking parts of the system. A proper test suite would solve this problem. Elaborating on this, a test suite would also make requirements for contributing more clear, as there are currently no guidelines that your pull request needs to comply with. Another thing that can be improved is the planning of the project. At this moment, it is not entirely clear what we can expect of Sonic Pi in the future. Milestones are not very actively used, so it would be a big improvement if it was clear which people are working on what features and what the planning of the project is in the long run.

References

1. Rozanski, Nick, and Eoin Woods. *Software systems architecture: working with stakeholders using viewpoints and perspectives*. Addison-Wesley, 2012.
2. Sonic Pi, http://sonic-pi.net/. Accessed on March 14, 2016.
3. samaaron/sonic-pi: The Live Coding Synth for Everyone, https://github.com/samaaron/sonic-pi. Accessed on March 14, 2016.
4. SuperCollider >> SuperCollider, https://supercollider.github.io. Accessed on March 14, 2016.
5. Overtone - Collaborative Programmable Music, http://overtone.github.io/. Accessed on March 29, 2016.
6. sonic-pi/HOW-TO-CONTRIBUTE.md at master - samaaron/sonic-pi, https://github.com/samaaron/sonic-pi/blob/master/HOW-TO-CONTRIBUTE.md. Accessed on March 30, 2016.
7. Introduction to OSC | opensoundcontrol.org, http://opensoundcontrol.org/introduction-osc. Accessed on March 26, 2016.
8. Raspberry Jessie description, https://www.raspberrypi.org/blog/raspbian-jessie-is-here/. Accessed on February 16, 2016.
9. Sonic Pi tutorial, https://www.raspberrypi.org/learning/sonic-pi-lessons/. Accessed on February 16, 2016.
10. WIPO magazine, *Sonic Pi: Getting Creative With Computer Programming*, http://www.wipo.int/wipo_magazine/en/2015/03/article_0007.html. Accessed on February 16, 2016.
11. Scintilla, http://www.scintilla.org/. Accessed on February 29, 2016.
12. Reek, https://github.com/troessner/reek. Accessed on February 26, 2016.
13. Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. FeatureIDE: An Extensible Framework for Feature-Oriented Software Development. Science of Computer Programming, 79(0):70-85, January 2014.
14. Unused gems? - Issue #928 - samaaron/sonic-pi, https://github.com/samaaron/sonic-pi/issues/928. Accessed on February 29, 2016.

Appendix A - Ruby Gems

The list of Ruby gems is quite long, and we did not manage to identify why each gem was used. Strangely enough, there are even gems present in the 'vendor' folder that are not used at all. These gems are scheduled to be used within a certain timespan [14]. There are also gems that are only used for testing. Among all the gems listed below, we can see that there are a few recurring categories: sound-related gems and gems related to parsing and compilation. These two categories are very important to the Sonic Pi system and it is not surprising to find them in the listed gems as well.

- ActiveSupport - a set of utility classes from the Rails framework
- alsa-rawmidi - access the alsa raw midi API via Ruby
- ast-2.0.0 - library for working with Abstract Syntax Trees
- atomic - provides a class for providing atomic updates
- benchmark-ips - can be used for benchmarking
- BlankSlate - provides an empty base class which can be used for e.g. dynamic proxies
- did-you-mean - provides 'did you mean suggestions' in errors
- ruby-ffi - ruby extension for programmatically loading dynamic libraries, binding functions within them, and calling those functions from Ruby code
- ffi-coremidi - Access the Apple Core MIDI framework API with Ruby.
- hamster - Efficient, immutable, and thread-safe collection classes for Ruby
- i18n - Ruby Internationalization and localization solution.
- interception - Interception (intercept + exception) allows you to intercept all exceptions as they are raised. (probably used for catching exceptions caused by a user's code)
- kramdown - kramdown is yet-another-markdown-parser but fast, pure Ruby, using a strict syntax definition and supporting several common extensions.
- metaclass - provides a `__metaclass__` method to all ruby objects
- midi-winmm - Realtime MIDI input and output with Ruby for Windows/Cygwin. Uses the WinMM system API
- midilib - midilib is a pure Ruby MIDI library useful for reading and writing standard MIDI files and manipulating MIDI event data.
- minitest - Ruby testing framework
- mocha - provides methods for stubbing and mocking
- multi_json - will simply choose the fastest available JSON coder
- narray - alternative, fast Ruby Array implementation
- parser - Parser is a production-ready Ruby parser written in pure Ruby
- parslet - Parslet makes developing complex parsers easy.
- Rouge - Pure Ruby syntax highlighter
- ruby-beautify - will pretty up ruby code.
- rubame - Rubame is a simple Ruby websocket game server.
- coreaudio - CoreAudio (Audio Framework of Mac OS X) wrapper library for ruby 1.9
- ruby-prof - ruby-prof is a fast code profiler for Ruby.
- rugged - Rugged is a library for accessing libgit2 in Ruby. It gives you the speed and

portability of libgit2 with the beauty of the Ruby language.

- Thread_safe - A collection of thread-safe versions of common core Ruby classes.
- unimidi - Platform independent realtime MIDI input and output for Ruby.
- wavefile - A pure Ruby gem for reading and writing sound files in Wave format (*.wav).
- websocket - Universal Ruby library to handle WebSocket protocol

Appendix B - Code Smells in the Sonic Pi system

Code Smell	Instances
Attribute	9
BooleanParameter	34
ClassVariable	16
ControlParameter	33
DataClumb	5
DuplicateMethodCall	365
FeatureEnvy	64
IrresponsibleModule	169
LongParameterList	23
NestedIterators	17
NilCheck	9
PrimaDonnaMethod	17
RepeatedConditionals	11
TooManyInstanceVariables	16
TooManyMethods	9
TooManyStatements	173
Uncommunicative (Parameter, Variable, Module)	397
UnusedParameter	31
UtilityFunction	166

TensorFlow™ - Open Source Library for Machine Learning Applications

By: Carmen Chan-Zheng, Ilse Verdiesen, Johan Carvajal-Godinez and Pranav Sailesh Mani
Software Architecture, Delft University of Technology

Abstract

TensorFlow™ is an open source software library developed by the Google Brain team for the purpose of conducting machine learning and deep neural networks research. The library performs numerical computation by using data flow graphs, where the nodes in the graph represent mathematical operations and the graph edges represent the multidimensional data arrays (tensors) which communicate between the nodes. The API has been used in the fields of medicine, translation services, and the analysis of financial markets. This chapter first describes TensorFlow™ by its features and stakeholders, secondly the architecture is analyzed by means of the context, development, and deployment view, and finally a conclusion is provided.

Table of Contents

1. Introduction

Machine learning is an artificial intelligence enabler that provides the foundations for extending the computer capabilities closer to human brain. To get there, new tools need to be developed, especially new models of computation that support faster application development cycle. This is not an easy task, which requires the help from the crowd sourcing community. TensorFlow™ is an extended version of DistBelief, a system developed for internal use at Google. DistBelief was internally used by Google employees to build large neural networks and scale training to thousands of cores in Google's datacenters. Some of the applications of Distbelief were image recognition, speech recognition, Google search etc. However, DistBelief had its own disadvantages. It was specifically designed for neural networks and was difficult to configure. It was also tightly coupled to Google's internal infrastructure. In order to overcome this problem, Google developed TensorFlow™ whose main purpose is to simplify real world use of machine learning systems. It is only recently (From December 18th 2015) that the TensorFlow™ team has started accepting contributions through GitHub before they were using the Gerrit system for collaborating. The history of developments starts from TensorFlow™ v0.5.0 till the current version 0.7.0. This chapter is intended to explore three key elements: (1) What is TensorFlow™, (2) its main architectures views, and (3) its more relevant perspectives. Following this purpose, this chapter has been organized to explore the stakeholders, the context view, the development view, the deployment view, the evolution-, the variability-, and the performance perspectives.

2. What is TensorFlow™?

TensorFlow™ is an open source library for developing machine learning applications. These applications are implemented using graphs to organize the flow of operations and tensors for representing the data. It offers an application programming interface (API) in Python, as well as a lower level set of functions implemented using C++. It provides a set of features to enable faster prototyping and implementation of machine learning models and applications for highly heterogeneous computing platforms.

2.1 Features

Tensorflow™'s webpage enlists the system's most important features, which are described as:

1. **Deep Flexibility**: Provides tools to assemble graphs for expressing diverse machine learning models. New operations can be written in Python and low-level data operators are implemented using in C++.
2. **True Portability**: Runs on CPUs, GPUs, desktop, server, or mobile computing platforms. That make it very suitable in several fields of application, for instance medical, finance, consumer electronic, etc.
3. **Connect Research and Production**: TensorFlow™ allows industrial researchers a faster product prototyping. It also provides academic researchers with a development framework and a community to discuss and support novel applications.
4. **Auto-Differentiation**: This is a key feature within the machine learning community. Gradient based machine learning algorithms benefit from automatic differentiation capabilities. As a TensorFlow™ user, you define the computational architecture for your predictive model, combine it with your objective function, and just add data to test your machine learning model.
5. **Language Options**: Python and C++. However, currently other APIs are being developed, for example a Ruby API.
6. **Maximize Performance**: Allows you to make the most of your installed hardware. Freely assign compute elements of your TensorFlow™ graph to different devices, and let TensorFlow™ handle the copies.

2.2 Stakeholders

TensorFlow™ is a very active community consisting of a very diverse group of Developers, Integrators, Researchers, Students, Architects, Software Engineers, Companies (i.e Engineers, managers, CEO, etc.), Consultants and Hardware Manufacturers (NVIDIA, ARM, Intel) (figure 1). Each group is assigned to each type of stakeholders according to the classification proposed in Mitchell et al.[1].

Type of Stakeholder	Description[1]	Application to TensorFlow™
Dormant Stakeholder (1)	The relevant attribute of a dormant stakeholder is power. Dormant stakeholders possess power to impose their will on a firm, but by not having a legitimate relationship or an urgent claim, their power remains unused.	Integrators
Discretionary Stakeholder (2)	Discretionary stakeholders possess the attribute of legitimacy, but they have no power to influence the firm and no urgent claims.	Researchers
Demanding Stakeholder (3)	Where the sole relevant attribute of the stakeholder-manager relationship is urgency, the stakeholder is described as "demanding." Demanding stakeholders, those with urgent claims but having neither power nor legitimacy, are the "mosquitoes buzzing in the ears" of managers.	Students, Consultants and Companies
Dominant stakeholders (4)	In the situation where stakeholders are both powerful and legitimate, their influence in the firm is assured, since by possessing power with legitimacy, they form the "dominant coalition" in the enterprise.	Hardware Manufacturers
Dangerous stakeholders (5)	We suggest that where urgency and power characterize a stakeholder who lacks legitimacy, that stakeholder will be coercive and possibly violent, making the stakeholder "dangerous," literally, to the firm.	Researchers using multiple libraries, for instance Microsoft CNTK
Dependent Stakeholder (6)	We characterize stakeholders who lack power but who have urgent legitimate claims as "dependent," because these stakeholders depend upon others (other stakeholders or the firm's managers) for the power necessary to carry out their will.	Architects and Software Engineers
Definitive Stakeholder (7)	By definition, a stakeholder exhibiting both power and legitimacy already will be a member of a firm's dominant coalition. When such a stakeholder's claim is urgent, managers have a clear and immediate mandate to attend to and give priority to that stakeholder's claim.	Developers

The following figure plots each of the stakeholder according to its type:

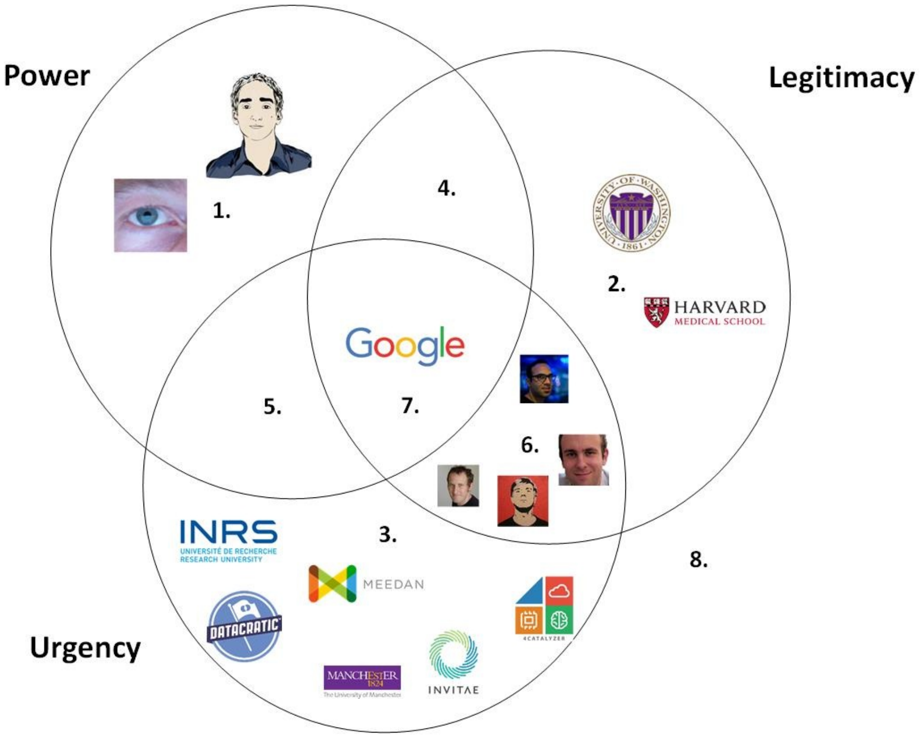


Figure 1. Stakeholder groups for TensorFlow

3. Architecture

The architecture of TensorFlow™ is described based on (1)*views*, which according to Rozanski and Woods[2] consists of elements or aspects of the architecture that are relevant to the concerns of the stakeholders, and (2)*perspectives* which are a set of related quality properties across a number of the system's architectural views that require consideration. In this paragraph first the context-, development- and deployment view are described, followed by the evolution-, variability- and performance perspective.

3.1 Views

In this paragraph the context view provides insight in the relationship between TensorFlow™ and its environment, the development view gives a description of models that are of concern of the stakeholders, and the deployment view shows the environment into which the system is deployed and the dependencies that the system has on its elements.

3.1.1 Context view

The purpose of this section is to elaborate on the relationships, dependencies and interactions between TensorFlow™ and its environment. The main goal is discovering how these elements affect TensorFlow's™ architecture. It will help to understand TensorFlow™

boundaries, as well as, its scope.

3.1.1.1 Scope definition

For this chapter, the most relevant elements surrounding TensorFlow™ have been considered. There were identified 5 key aspects that enables the operation of the library. These are: stakeholder community, repository management, execution platforms, visualization tools, and Library dependencies. These elements are described in more detail in the following subsections.

3.1.1.2 Stakeholder community

In the previous section the stakeholder community was described in detail. From this analysis, four key players were identified: developers, researchers, integrators and companies. Developers are those who enable new models, operations and tools within the API. They have a strong background in computer science. On the other hand, researchers use the latest version of TensorFlow™'s implementation to develop applications and study machine learning algorithms. Integrators are in charge to make decisions on whether or not a contribution can be integrated to the code base. These contributions are being proposed mainly by developers and researchers. Finally, the companies explore the results of researchers and try to enable new features on their products, taking advantage of their improvements. All of these four actor define the contributor community of TensorFlow™, which is supported by Google, in order to provide resources, but to gain expertise that could be used in future products.

3.1.1.3 Repository Management

TensorFlow™ is hosted at GitHub under the Google's support. As mentioned previously, integrators are in charge of analyzing pull requests and determining if these can be merged. TensorFlow™ uses Jenkins as a continuous integration platform. Also, Docker is used to support the their delivery process.

3.1.1.4 Execution Platforms

The execution environment surrounding TensorFlow™ is very heterogeneous. The Library supports ports for CPU, GPU and mobile platforms. It makes the library suitable for many application fields, but increases the implementation complexity. For example, for GPU implementation, currently only CUDA is supported, and for mobile platforms only Android. Also, there is a price to be paid when offering flexibility, which is performance. Later in this chapter we will see an analysis of this aspect.

3.1.1.5 Visualization tools

Visualization is a key element for researchers. It allows to compare, but also to play with parameters to optimize the algorithm performance. Tensorboard is a module provided for that purpose. It takes the results from execution and enables the user to display, and compare different runs.

3.1.1.6 Dependencies

As many other libraries, TensorFlow™ requires features from other systems, not just for maintaining the repository, but for enabling new functionalities. The main dependencies are the NVIDIA driver support for CUDA, the Python and C++ language support. There are several other dependencies that would be detailed in the development view.

In figure 2 the relation between the surrounding elements and the TensorFlow™ library are illustrated.

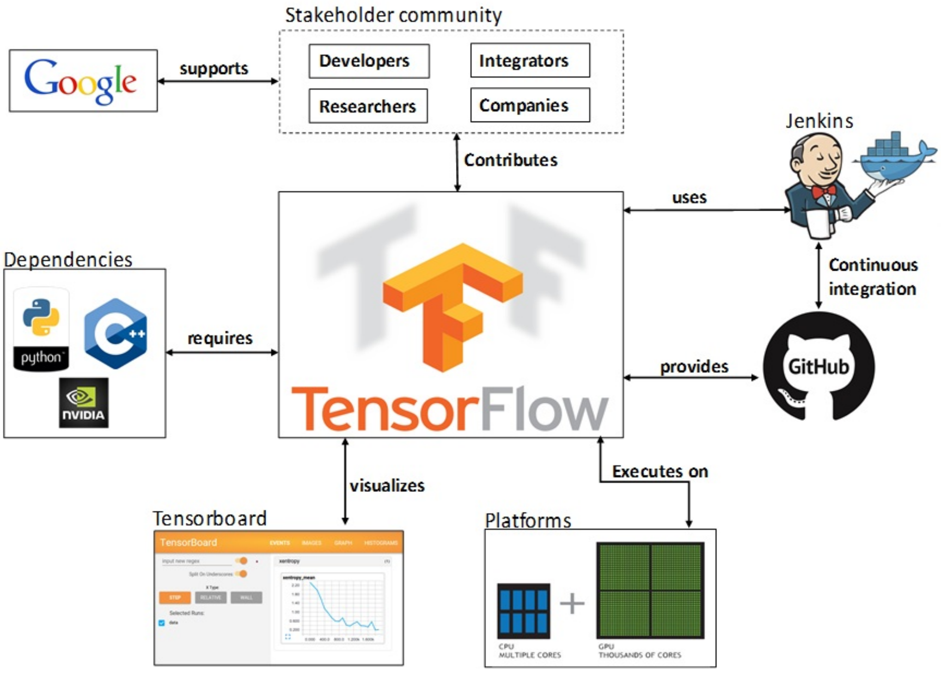


Figure 2 Context diagram for TensorFlow™

3.1.2 Development view

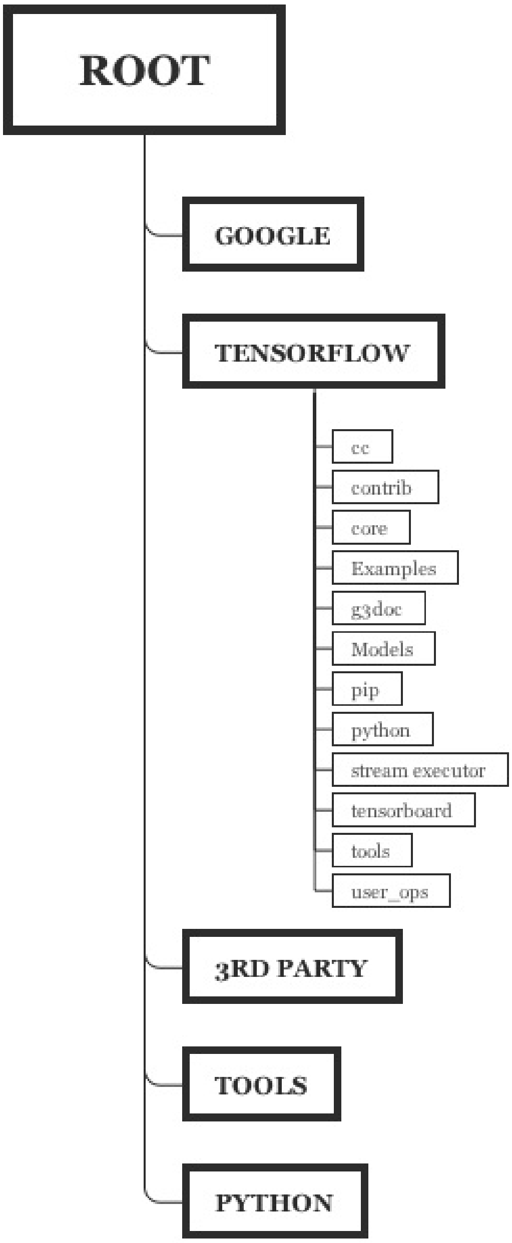
This view addresses the specific concerns of the software developers and testers[2]. This section contains the description of three different models: Codeline Models, Module Structure Model, and Common Design Models.

3.1.2.1 Codeline Models

This section explores the code structure of TensorFlow™ with the purpose of getting a better understanding on how the project is organized.

3.1.2.1.1 Source code hierarchy

TensorFlow™ 's root directory at GitHub is organized in five main subdirectories: google, tensorflow, third-party, tools and util/python. Additionally, the root directory provides information on how to contribute to the project, and other relevant documents. In figure 3, the source code hierarchy is illustrated.



```
Figure 3. TensorFlow™ source code organization
```

Following subsections will elaborate on the purpose of the five main subdirectories shown in figure 3:

3.1.2.1.2 Google

This subdirectory provides an instance of ¨Protocol Buffers¨ library, a Google's language and platform neutral, mechanism for serializing structured data. That is required since TensorFlow™ supports implementations in C++ using a single Python API, which requires a common data interface. Also, it enables support to implementation with other programming languages, for instance, Java, Ruby and many others.

3.1.2.1.3 Tensorflow

This is the heart of the library's implementation. It contains a set of subdirectories intended to:

- cc: declaring function wrappers for C++ code.
- contrib: containing features and contributions that eventually should get merged into tensorflow/core.
- core: providing an implementation for the main functionalities of TensorFlow™.
- examples: containing reference applications for contributors.
- g3docs: contains documentation for Python and C++ APIs.
- models: containing models for specific application implementation.
- python: containing the implementation of Python to support the application development API.
- stream-executor: providing an interface with hardware accelerators (GPUs).
- tensorboard: containing a suite of web applications for inspecting and understanding of TensorFlow™ runs and graphs.
- tools: providing a proper execution environment for the library.
- user_ops: providing wrappers for customized user functions (i.e. ackermann).

3.1.2.1.4 Third-party

This directory provides instances of useful third-party libraries to help the TensorFlow™ application developers. Three main libraries are identified here:

- Eigen3: a C++ template library for linear algebra operations.
- gpus: a crosstool wrapper for compiling CUDA programs.
- [numpy] (http://www.numpy.org/): a package for scientific computing with Python.

3.1.2.1.5 Tools

This provides a place to put a script to indicate Bazel which version of Python is being used.

3.1.2.1.6 Util/ python

This provides a place to put a script to generate relative paths for Python, so it works in both a local or remote repository.

3.1.2.2 Module Structure Models

The purpose of this section is to focus on the organization of the TensorFlow™'s repository and group the modules into layers of abstractions.

3.1.2.2.1 Classify the Modules

As shown in the section "Codeline Models", the TensorFlow™ GitHub Repository is organized and classified as shown in Figure 3, these can be classified in the following way:

- Application development modules: Python API (*pp* folder), C++ API(*cc* folder), Tensorboard, contributions (*contrib* folder), examples, tools such as Jenkins, pip and Docker (*tools* folder in root directory) and useful documentation such as: APIs documentation, tutorial, getting started example located in *g3docs* directory.
- Framework modules: Models, user operations, then from *tensorflow/core* directory: libraries (*lib* folder), and from the *root* directory: 3rd party.
- Platform modules: from *tensorflow/tensorflow* directory: Stream executor and the _*tensorflow/tensorflow/core* directory: kernels implementation, common_runtime, distributed_runtime, etc.

3.1.2.2.2 Module Dependencies

To identify the module dependencies, each code from the application development, framework and platform modules group were analyzed. For example, the *function.py* (from the Python API of the Application development group) shows a clear dependency to the Platform modules and Framework modules. This same approach was applied for each source file from the repository and the main identified dependencies are:

- Application development group and Framework group share dependencies between them.
- Application development group with Platform group.
- Platform group and Framework group share dependencies between them.

3.1.2.2.3 Layering Rules

In *Classify the Modules* section, the modules have been classified into different groups. These groups can be used as the layers to organize the structure of the TensorFlow™ repository:

1. Application Development layer: This layer contains the tools for the user to develop any TensorFlow™ application. It contains API documentations, examples, tutorials, contributions from other users, installation and distribution tools.
2. Framework layer: This layer contains the definition of the operations used for the application development, models for specific application implementation and session

implementations.

3. Platform layer: This layer provides the interface with hardware (GPUs and CPUs), operation kernels, platform operations, datatype definition and execution framework.

The system's layer organization is shown in the following figure:

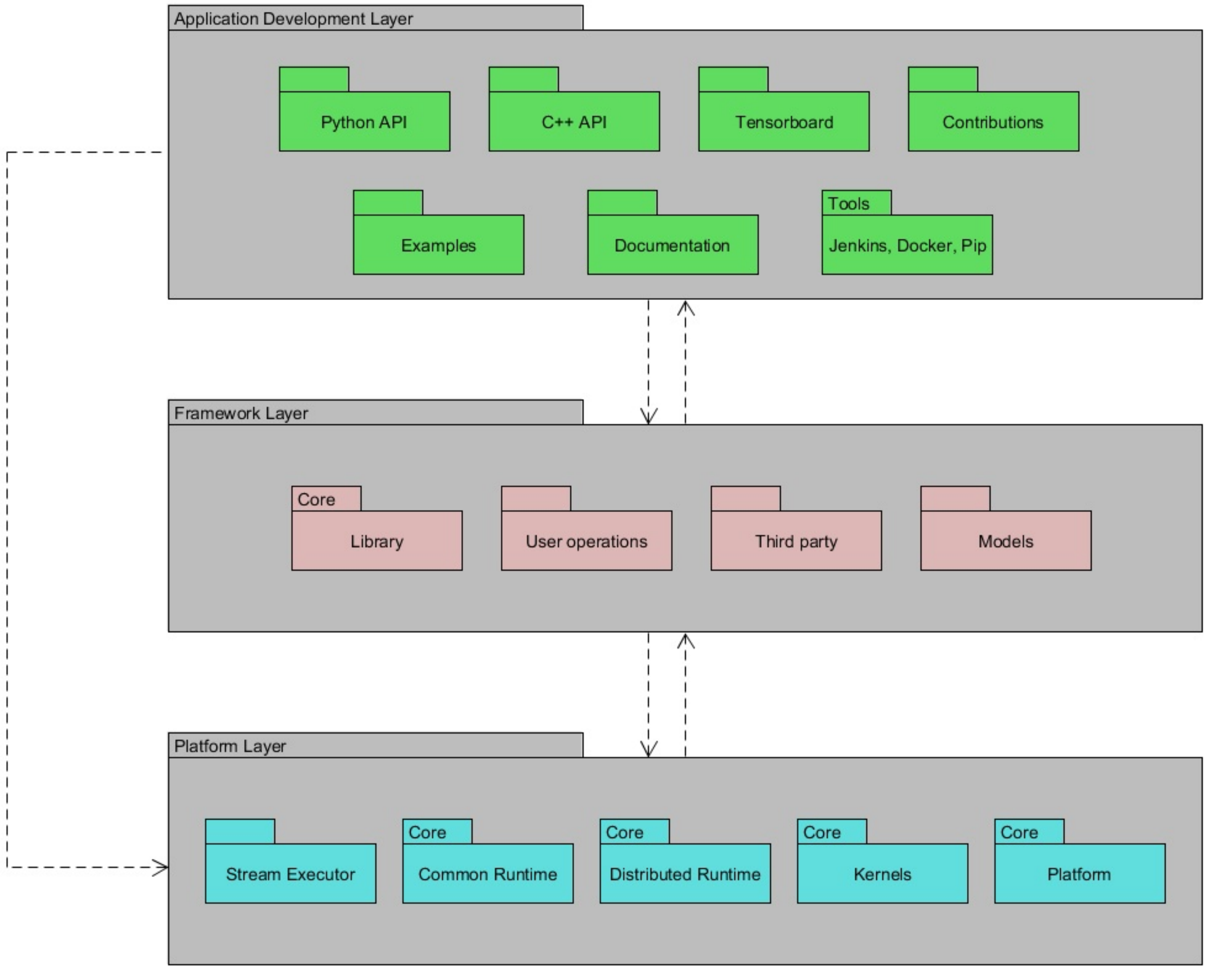


Figure 4. Structure model of the TensorFlow™

3.1.2.3 Common Design Models

From the structure obtained in figure 4, we started to identify modules which are common processing.

3.1.2.3.1 Common Processing elements

The following elements can be identified as common processing elements:

- Message logging and instrumentation: For logging error messages in Python and C++ it is possible to log utilities. This controls which methods from pyglib.logging are used and a logline prefix using the google2 format is assembled.
- Use of third-party libraries: Third-party libraries that are used are CUDA, Eigen, NumPy

and protobuf.

3.1.3 Deployment view

According to Rozanski and Woods[2], the deployment view describes the environment into which the system is deployed and the dependencies that the system has on its elements. This view encapsulates the hardware environment that a system requires, the technical environment requirements for each element, and the mapping of the software elements to the runtime environment that will execute them. The next two sections will discuss the overview of TensorFlow™ Runtime Requirements, and later, it presents two deployment view models.

3.1.3.1 Overview of TensorFlow™ Runtime Requirements

As discussed in the previous section, TensorFlow™ provides Python API and C++ API to the users to express a variety of algorithms, such as something as simple as math computation to a variety of machine learning models. In order to deploy (or distribute) the system, the TensorFlow™ team has provided four methods to install the Python API:

1. Clone the files from GitHub source (For Linux 64-bit and Mac OS X 64-bit): This method might be the most tedious method since the user needs to install all dependencies separately. It requires Bazel, which is a build tool that builds code quickly and reliably. Then it requires to install all the Python dependencies, and lastly, (if needed) it requires the installation of the CUDA package. Less experienced users can follow the next three methods.

2. Pip install (For Linux 64-bit and Mac OS X 64-bit): pip is a package management system used to install and manage software packages written in Python. This method might upgrade previous installed Python packages.

3. Virtual env install (For Linux 64-bit and Mac OS X 64-bit): a tool that allows the creation of isolated Python environments. Using this, TensorFlow™ will be installed in its own directory.

4. Docker (For any Operating System 64-bit environment): tool which wraps up a piece of software into one container where contains everything needs to run the system. This tool allows the use of TensorFlow™ in different operating system environments.

On the other hand, the C++ API deployment is only available through cloning the files from GitHub source. Without regard of which Operating System is currently been used, the system needs Third-party software requirements, such as Python 2.7 or Python 3.3+

installed in the system. Also, the GPU version only can be run in Linux system and it requires the installation of Cuda Toolkit (at least version 7.0) and cuDNN (at least version v2).

Lastly, as for the hardware requirement, the TensorFlow™ team has not specified the minimum hardware requirements for running its architecture. However, since it uses the CUDA toolkit, we can assume that this is only intended for NVIDIA's GPU. Beside this assumption, the developer team only mentions that it will run in a conventional desktop CPU and optionally in GPU (in TensorFlow™ terminology CPU and GPU are called as devices). It can be deployed to a single device environment, multiple device environment (several devices within the same machine) or distributed environment (devices are distributed in several machines). However, there are examples that a written application using the Python API has been successfully deployed to an Android device by using Bazel, Android NDK and Android SDK (Click here for further information).

3.1.3.2 Deployment View Models

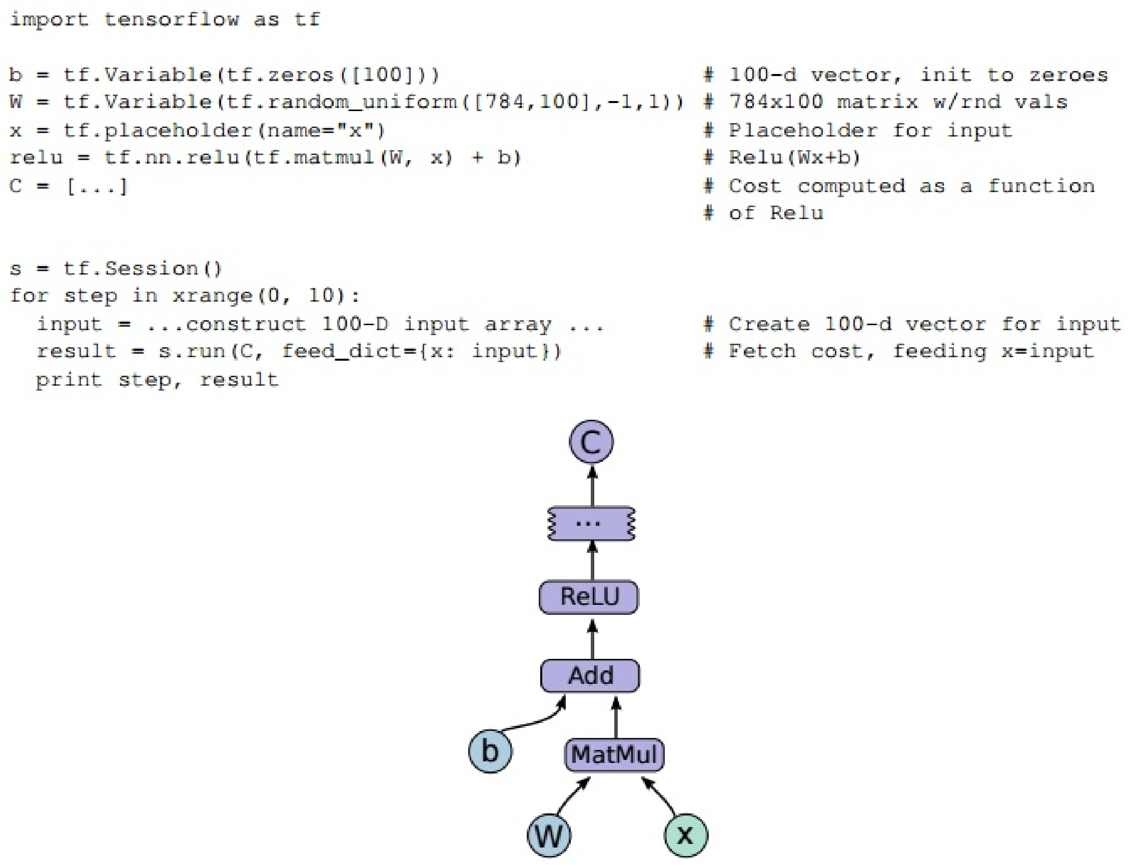
According to Rozanski and Woods[2], there are three models to describe the architecture of TensorFlow™ from the deployment view:

- Runtime Platform Models
- Network Models
- Technology Dependency Models

The next two subsections will detail the first two models since these are the most relevant for TensorFlow™.

3.1.3.3 Runtime Platform Models

Rozanski and Woods[2] state that the Runtime Platform model is the core of this view. It defines the set of hardware nodes required, the interconnection within the node, and software elements hosted in the hardware nodes. In order to build this model, it is important to understand how the system underlying mechanism works. According to the whitepaper[3], a user creates an algorithm in TensorFlow™ with the provided API and internally that algorithm is described by a graph that represents a dataflow computation. In the graph, each node corresponds to an operation (for example: an arithmetic operation), the values that flow along the edges are tensors (that's why it is called TensorFlow™!). The following figure obtained from the whitepaper[3] shows an example of TensorFlow™ code fragment with its corresponding computation graph:



```
import tensorflow as tf

b = tf.Variable(tf.zeros([100]))                   # 100-d vector, init to zeroes
W = tf.Variable(tf.random_uniform([784,100],-1,1)) # 784x100 matrix w/rnd vals
x = tf.placeholder(name="x")                       # Placeholder for input
relu = tf.nn.relu(tf.matmul(W, x) + b)             # Relu(Wx+b)
C = [...]                                          # Cost computed as a function
                                                   # of Relu

s = tf.Session()
for step in xrange(0, 10):
  input = ...construct 100-D input array ...       # Create 100-d vector for input
  result = s.run(C, feed_dict={x: input})          # Fetch cost, feeding x=input
  print step, result
```

```
                         Figure 5. TensorFlow™ code fragment with its computation
graph example[[3]]
```

The whitepaper[3] also mentions that the main components in TensorFlow™ are the clients, the master and one or more worker processes. If a user wants to run an algorithm, the user as a client, interacts with the master where it runs a placement algorithm to decide how to distribute the computation among all the worker processes. Each worker processes is responsible for arbitrating access to one or more computational devices. Each of the devices can communicate to each other through Send- and Receive node functions of the TensorFlow™ library.

The following figure shows the runtime model for TensorFlow™:

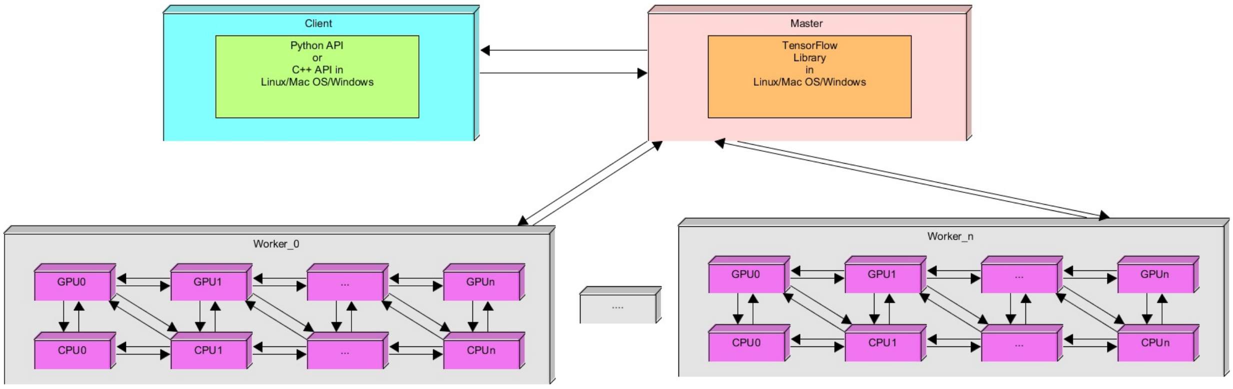


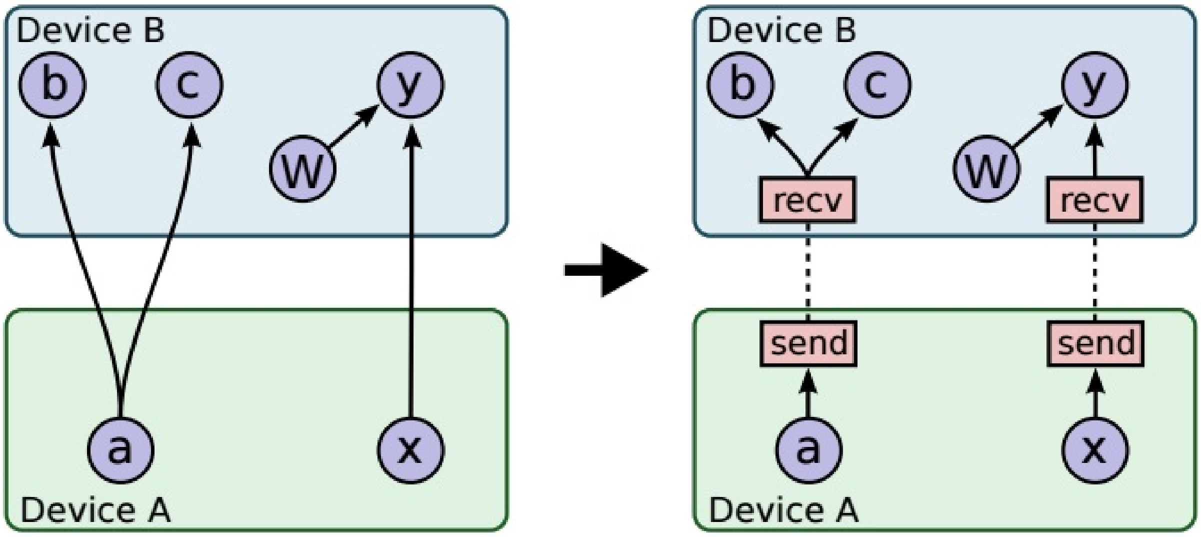
Figure 6. Runtime Platform Model

The model above applies for single, multiple and distributed environment.

3.1.3.4 Network Models

The Runtime Platform Model describes a high level of communication between the client, master and worker. The Network model describes specifically which nodes need to be connected and any other specific services and bandwidth requirements. In TensorFlow™ this model is applied to the cross device communication. Cross device communication means data transference between two or more devices (GPUs or CPUs), in which each device corresponds to a node in the network model.

According to the whitepaper[3], once the master has run the placement algorithm, the graph is partitioned into a set of subgraphs, one per device, as shown in the left graph of the following figure:



```
                                Figure 7. Network Model: TensorFlow™ Cross Device Commun
ication
```

Once it is placed, TensorFlow™ removes the graph x -> y and replaces it by two subgraphs: x -> send node and receive -> y as shown in the right side of Figure 7. At runtime, the implementation of send and receive nodes (which is part of TensorFlow™ library) coordinate the transfer of data across devices. By using this implementation, it isolates all communication inside Send- and Receive nodes which simplifies the rest of runtime. Also, one of the benefits of this implementation is that it decentralizes the system: the sender and receiver nodes impart the necessary synchronization between different workers and devices while the master only needs to issue requests per graph execution to each worker rather than being involved in the scheduling of cross device communication, thus, it makes the system much more scalable.

3.2 Perspectives

In the following paragraphs the evolution perspective gives insight in the history of TensorFlow™, the variability perspective shows the modular development and different configurations, and the performance perspective gives an overview of the performance of TensorFlow™ compared to other machine learning applications.

3.2.1 Evolution perspective

The history of developments starts from TensorFlow™ v0.5.0 till the current version 0.7.0. In order to know the history of developments, the roadmap of TensorFlow™ was first looked at. Currently they are working on the support for iOS support and OpenCL support. The initial version of distributed support is available in the source code. However, this has not been released in the binary versions as of now.

3.2.2 Variability perspective

Many of the libraries in TensorFlow™ were developed separately and then integrated into the main codebase. This kind of modular development made it very easy to implement variability in TensorFlow™. TensorFlow™ can be run in two different configurations, namely single device execution and multi-device execution. Apart from this, there is a GPU version of TensorFlow™ and non-GPU version of TensorFlow™. It is supported in two separate Operating Systems, Linux and Mac OS. So, TensorFlow™ has a lot of variability built into it and the modular development of code is one of the reasons why TensorFlow™ can have this high amount of Variability. The core library is written C++ along with CUDA and for running the code on a machine containing a NVidia GPU. CUDA must be installed on the computer so as to support the code written for the GPU. In order to discuss the strategy used for implementing variability, we must first discuss the components of the TensorFlow™ application. The components in the TensorFlow™ application [3] are:

3.2.2.1 Client

The client is the application written in C++/ Python that interacts with the TensorFlow™ libraries and perform the required operations. The client creates a "Session" for interacting and using the TensorFlow™ libraries.

3.2.2.2 Master processes and Worker processes

The session created by the client interacts with the Master process, which in turn splits the client program into different sub programs/ sub graphs and allocates the work among different Worker Processes. These worker processes are responsible for running the computations on each of the devices/ processors. The devices in this case maybe GPU or CPU.

We can see the modularity built into the application. This kind of modularity allows for a huge amount of variability where each process works independent of each other. For example, if we want TensorFlow™ to run locally on a single computer, the client master and the worker processor all run on a single machine in the context of a single Operating system. This single computer can either be with or without the GPU. This is managed by the worker processes. So, this handles with variability of processors. The difference between the local and distributed versions is that they share the same code but the difference is that the client, the workers and the master can all be on different machines. These different tasks are containers in jobs managed by a cluster scheduling system.

For distributed versions of TensorFlow™, gRPC is used for interprocess communication. Detailed steps for setting up the distributed version TensorFlow™ can be found here. We have identified the relevant steps from the source and have listed them below. During build time, we must specify whether we need a CPU only or the GPU version. We need to have bazel installed on the computer in which we are building and currently this can be done only through source based installation of TensorFlow™.

3.2.3 Performance perspective

According to Rozanski and Woods [2], some of the important concerns are response time, throughput, turnaround time, scalability, predictability, Hardware Resource Requirements and Peak Load Behaviour. In our chapter, we focus only on Response time and throughput. We don't go into detail on these aspects, but we will be discussing at a high level about Performance Perspective of TensorFlow™. Performance Perspective is very important to a tool like TensorFlow™ as the performance of machine learning applications written in

TensorFlow™ is extremely crucial. In complex machine learning applications, the best hardware might not give us the best results and the performance mainly depends on the code quality and the effectiveness with which memory is utilized by the software. Due to the complexity of the algorithms involved it is an extremely difficult task to benchmark the performance of TensorFlow™ by ourselves. So, we decided to use the information provided by other developers involved in the field of machine learning. These benchmark details were obtained from this document in GitHub. Convnet has been used for the benchmarking. Convolutional network (Convnet) is a specific artificial neural network topology that is inspired by biological visual cortex and tailored for computer vision tasks. The benchmarks were performed on a machine with the following specifications: `6-core Intel Core i7-5930K CPU @ 3.50GHz` + `NVIDIA Titan X` + `Ubuntu 14.04 x86_64` . Some of the popular imagenet models were picked up and benchmarked. The benchmarking results are also done with other libraries available and we can draw some conclusions and compare them with other libraries to get a good idea of the actual performance of TensorFlow™ and whether it is one of the best libraries for machine learning. The benchamarked results are tabulated below:

AlexNet (One Weird Trick paper) - Input 128x3x224x224

Library	Class	Time (ms)
CuDNN[R4]-fp16 (Torch)	cudnn.SpatialConvolution	71
Nervana-neon-fp16	ConvLayer	78
CuDNN[R4]-fp32 (Torch)	cudnn.SpatialConvolution	81
Nervana-neon-fp32	ConvLayer	87
fbfft (Torch)	fbnn.SpatialConvolution	104
TensorFlow	conv2d	151
Chainer	Convolution2D	177
cudaconvnet2*	ConvLayer	177
CuDNN[R2] *	cudnn.SpatialConvolution	231
Caffe (native)	ConvolutionLayer	324
Torch-7 (native)	SpatialConvolutionMM	342
CL-nn (Torch)	SpatialConvolutionMM	963

Overfeat [fast] - Input 128x3x231x231

Library	Class	Time (ms)		
CuDNN[R4]-fp16 (Torch)	cudnn.SpatialConvolution	242		
CuDNN[R4]-fp32 (Torch)	cudnn.SpatialConvolution	268		
fbfft (Torch)	SpatialConvolutionCuFFT	342		
TensorFlow	conv2d	349		
Chainer	Convolution2D	620		
cudaconvnet2*	ConvLayer	723		
CuDNN[R2] *	cudnn.SpatialConvolution	810		
Caffe	ConvolutionLayer	823		
CL-nn (Torch)	SpatialConvolutionMM	963		
Caffe-CLGreenTea	ConvolutionLayer	2857	616	2240

OxfordNet [Model-A] - Input 64x3x224x224

Library	Class	Time (ms)		
CuDNN[R4]-fp16 (Torch)	cudnn.SpatialConvolution	471	CuDNN[R4]-fp32 (Torch)	cudnn.SpatialCo
Chainer	Convolution2D	885		
TensorFlow	conv2d	982		
fbfft (Torch)	SpatialConvolutionCuFFT	1092		
cudaconvnet2*	ConvLayer	1229		
CuDNN[R2] *	cudnn.SpatialConvolution	1099		
Caffe	ConvolutionLayer	1068		
Torch-7 (native)	SpatialConvolutionMM	1105	350	755
CL-nn (Torch)	SpatialConvolutionMM	3437	875	2562
Caffe-CLGreenTea	ConvolutionLayer	5620	988	4632

GoogleNet V1 - Input 128x3x224x224

Library	Class	Time (ms)
CuDNN[R4]-fp16 (Torch)	cudnn.SpatialConvolution	462
CuDNN[R4]-fp32 (Torch)	cudnn.SpatialConvolution	470
Chainer	Convolution2D	687
TensorFlow	conv2d	905
Caffe	ConvolutionLayer	1935
CL-nn (Torch)	SpatialConvolutionMM	7016
Caffe-CLGreenTea	ConvolutionLayer	9462

TensorFlow™ and Chainer are benchmarked with CuDNN, but it is not explicitly mentioned, and one might think that these frameworks as a whole are faster, than for example Caffe, which might not be the case. However, one thing that can be inferred from the table above is that TensorFlow™ isn't the best library out there and it as its own set of performance issues and there is a lot of room for improvement. TensorFlow™ only uses cuDNN v2 and its performance is almost 1.5x slower than Torch with cuDNN v2. It must be noted that the benchmark results are done using only a single-powerful GPU. However, these benchmarks were run on an older version of TensorFlow™ and the current benchmarking results are not yet available. More detailed and a clear view of TensorFlow™'s performance results for all kinds of architectures and number of machines can be expected in the next versions of whitepaper. However, not much detail is available from Google itself. Benchmarking performance is on their roadmap and more clear results can be expected in the future.

4. Conclusion

Based on the analysis of the architecture, we can conclude that TensorFlow™ is a startup project which is not yet mature and has a lot of room for future improvements. In the previous section it was shown that TensorFlow™ has performance issues compared to other machine learning applications. Next to this, the variability of TensorFlow™ is limited at this moment and it would be beneficial to extend support to OpenCL and add different platforms, such as iOS. Despite these limitations, we believe that TensorFlow™ has a lot of potential. Opening up TensorFlow™ to the OpenSource community is a very important step to tackle these limitations. When Google can address the current shortcomings it might very well become the defacto standard for machine learning.

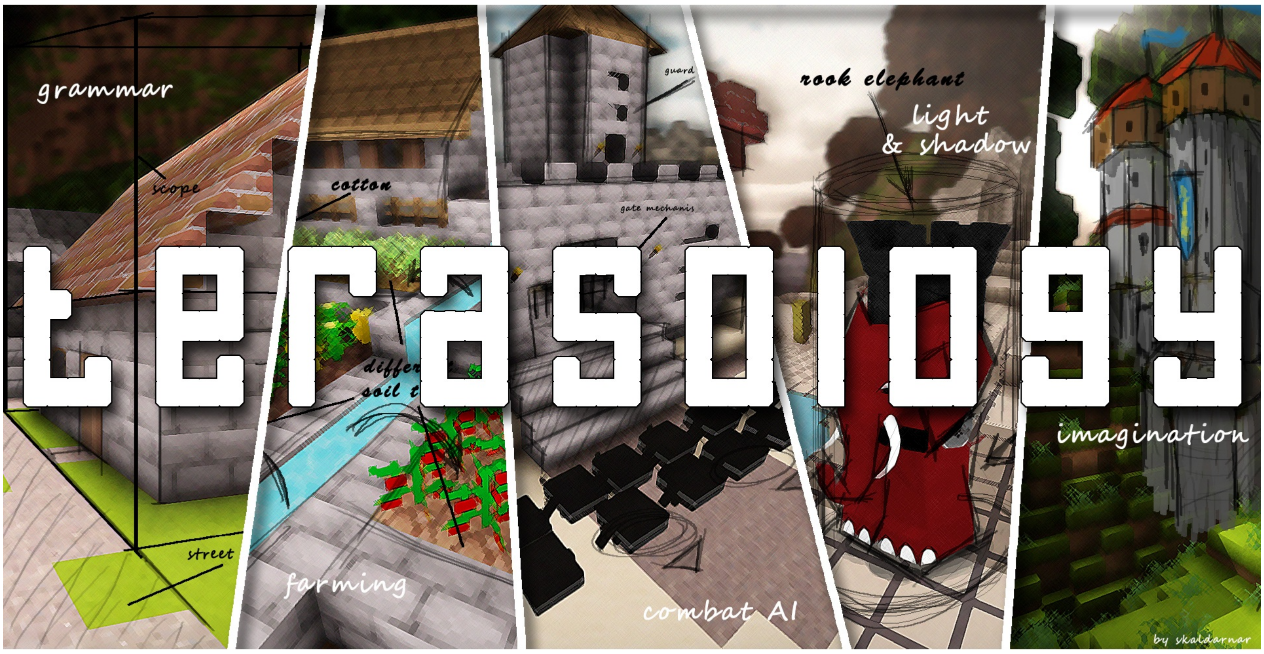
5. References

[1] Mitchell, R. K., Agle, B. R., & Wood, D. J. (1997). Toward a Theory of Stakeholder Identification and Salience: Defining the Principle of Who and What Really Counts. The Academy of Management Review, 22(4), 853–886. Retrieved from http://www.jstor.org/stable/259247

[2] Nick Rozanski and Eoin Woods. (2011). Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives. Addison-Wesley Professional.

[3] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Rafal Jozefowicz, Yangqing Jia, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Mike Schuster, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

Terasology: open source game development project



By Bas de Böck, Alexander Grooff, Arkka Dhiratara and Jan Zegers

Delft University of Technology

Abstract

Terasology is a video game similar to Minecraft. It started as part of a research project, but soon evolved into an active open source project. In this chapter, Terasology is described by different views and perspectives. On the one hand, focus is given to the development view, because Terasology currently finds itself in the development phase. On the other hand, focus is given to the variability perspective, as the modular architecture is the thing that makes the project special. Related to the next phase of the project, a description is given of the deployment view and performance and scalability perspective. The results from this research can provide new contributors to gain an architectural insight into the Terasology project, enabling them to more easily understand the project and contribute to it.

Table of Contents

1. Introduction

Terasology is a video game inspired by Minecraft [1]. It started as a research project, but soon evolved into a serious open-source project. Terasology is developed by a small, but warm and active community, which we have experienced to be welcome and friendly to newcomers.

Terasology is an interesting project as subject of study for software architects, as the community behind it is trying to get from the development phase to the deployment phase. In this document, we describe the different architectural views and perspectives of the Terasology project. We start with a stakeholder analysis, which gives insight into the people that are involved. After that, the context view is given to place Terasology within its environment. Next we describe the development view, as the project is still in its development phase we will give extra attention to this part. Because modularity is a key element of the project, the following variability perspective gets extra attention too. As previously explained the Terasology project is transferring to the deployment phase, as such this will be described in the deployment view. At last, we provide a description of the scalability perspective as extra metric on the maturity of the project. We end this chapter with our conclusion on the Terasology project architecture.

2. Stakeholders

Terasology originally started under the name of Blockmania by Benjamin "begla" Glatzel, and was meant as a research project for procedural terrain generation and efficient rendering techniques in Java using the Lightweight Game Java Library (LWGJL) [2]. The first two members to join the project where Anton "small-keeper" Kireev and Rasmus "Cervator" Praestholm, which lead to the starting point of the conversion of a research project demo to a full-fledged game.

Terasology’s goal is to stake out its own niche by adding gameplay aspects from NPC-helper and caretaker games as Dwarf Fortress and Dungeon Keeper, yet paying ample tribute to Minecraft in look and origin. This results in a sandbox game with creative and building aspects that enable players to build constructions out of textured cubes in a 3D procedurally generated world. Based on the activated modules, the game provides activities as exploration, resource gathering, crafting and combat; as well as gameplay modes such as a survival mode where a player must acquire resources to build the world and stay alive. This module structure allows modders to come up with creative ideas as it allows them to create and ship a mod with a few simple commands, making it immediately available for download in-game by the users.

The main developers "Cervator" and Martin "msteiger" Steiger are the ones that open almost every issue. The rest of the issues have been opened by users of the system. We found that the four top contributors consist out of: "Cervator", "msteiger", Malo "MaloJaffre" Jaffre and Josh "Josharias" Zacharias [3].

"Cervator" can be seen as the main integrator of the Terasology project, as he acts as a guardian for the project’s quality, while at the same time keeping contributions "in-flight" by communicating modification requirements to the original contributors. When a pull request is made which lays outside of his skillset, he assigns it to the developer that is specialized in that part of the system. The developers decide in consultation with the contributors whether to accept a contribution. In this project, three main factors leading to acceptance arise, namely: code quality, code style and project fit. The time it takes to make the acceptance decision mostly depends on the reviewer availability, responsiveness of the contributors and the overall complexity of the code. The quality of the code is evaluated by looking at the understandability, documentation and the added value to the project. For continuous integration, Terasology uses Jenkins as a tool to evaluate quality.

By using the knowledge we gained from the analysis of the Terasology Github page, forum and social media pages we were able to create the following stakeholder overview as can be seen in table 1.

Type	Entities
Developers	**Construct and deploy the system. The following developers have been active in the past year and have been divided per team.**
	Architects: Benjamin "begla" Glatzel, "Immortius", Martin "MarcinSc" Sciesinski, "shartte", "flo", "emanuele3d"
	Design Team: "Cervator"
	General: Linus "LinusVanElswijk" van Elswijk, Jakub "Limeth" Hlusicka, "unpause", "OvermindDL1", "prestidigitator", Gustavo "gtugablue" Silva, Flavio "sk0ut" Couto, "Netopya", Andre "andrelago13" Lago , "MaloJaffre"
	GUI Team: Piotr "Halamix2" Halama
	World Team: Tobias "Skaldarnar" Nett, "msteiger", "Josharias"
Users	**The users of Terasology consist out of players from all over the world. This is depicted by the activity on their social channels.**
	4k+ likes on Facebook
	400 subscribers on Reddit
	1200+ Twitter followers
Suppliers	**Build and/or supply the hardware, software, or infrastructure on which the system will run.**
	LWGJL
	Github
	Inspired by games like Minecraft, Dungeon Keeper and Dwarf Fortress
Assessors	**Oversee the system's conformance to standards and legal regulations.**
	"Cervator" by enforcing the use of an Apache 2.0 license agreement and Terasology's Code of Conduct.
Support Staff	**Support is being provided via Github, Twitter, Facebook, Reddit and the Terasology Forum.**
	"Cervator", serves as main moderator on the support sites.

Table 1: Stakeholder Overview

The power interest grid, as depicted in figure 1, provides a prioritization of the stakeholders by their power/interest ratio.

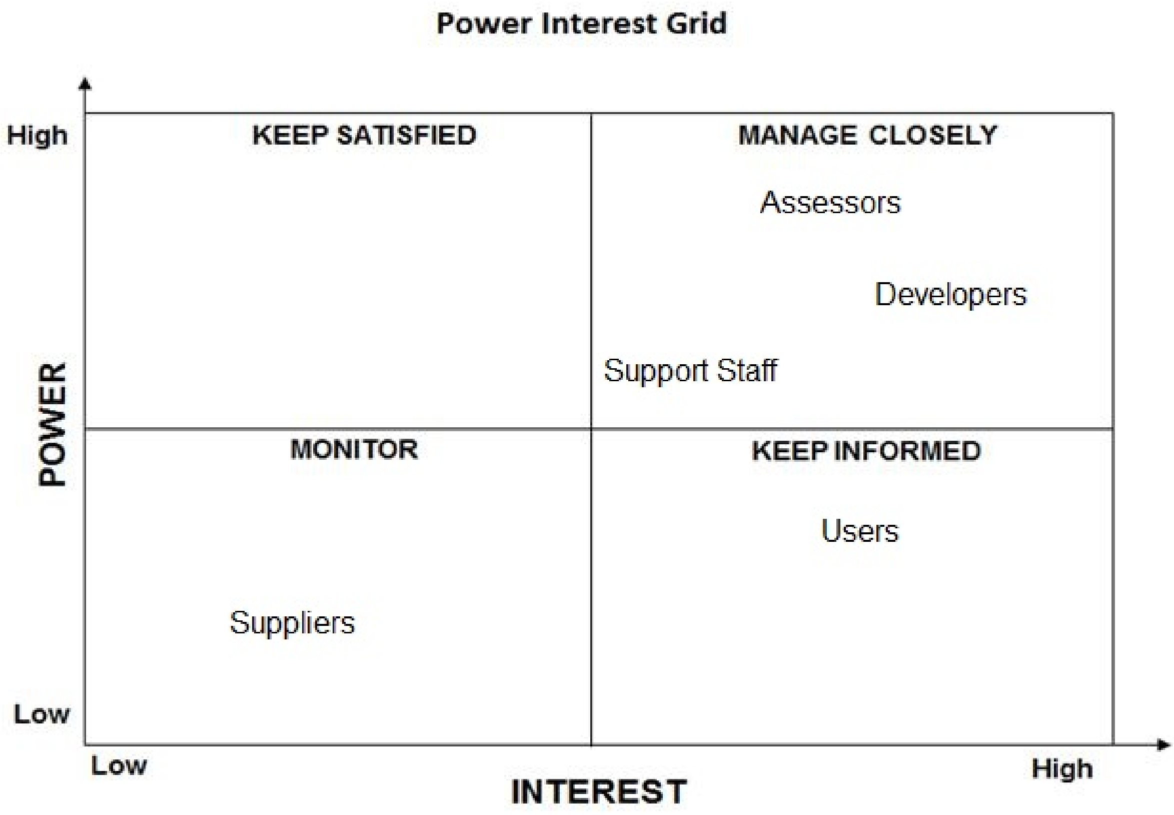


Figure 1: The stakeholders in a power/interest grid

3. Context view

In order to provide a better understanding of the Terasology architecture, we have identified the relationships, dependencies and interactions between entities in the Terasology project. In this context view, we have grouped the entities into stakeholders, development, community, external entities, and licenses, which can be seen in figure 2.

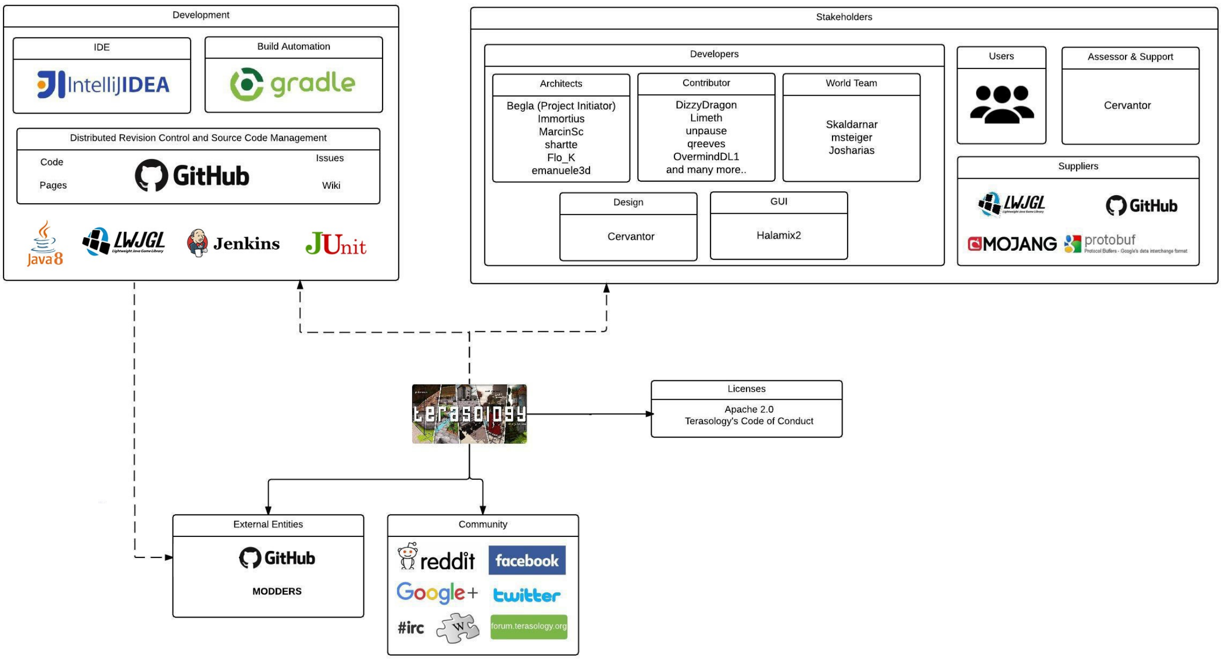


Figure 2: Context view showing the basic components of which Terasology consists of

Development

The development group consists out of the technical related entities in the Terasology project. Terasology is build using the Java programming language to deliver a cross-platform solution to their users. Dependent on that programming language, Terasology uses a native game library for Java which is the LWGJL. LWGJL provides developers access to graphic (OpenGL), audio (OpenAL) and parallel computing (OpenCL) development tools. Terasology's community prefers to use IntelliJ IDEA as their recommended Integrated Development Environment (IDE), though there is also support for other IDEs such as Eclipse and NetBeans. A project with this scale obviously depends on existing libraries, therefore, Terasology uses Gradle as their dependency management tool for monitoring the build process. Moreover, Terasology encourages the open source community on Github to contribute to the project. Github provides distributed revision control and source code management that enable developers and contributors to work together collaboratively over the Internet. Therefore, Terasology is also required to verify each contribution from the community. Jenkins supports this task by providing continuous integration and continuous delivery. It is able to provide a number of different required tests, such as JUnit testing, Checkstyle and PMD.

Community

As a project that aims at a wide range of users, Terasology is also required to maintain their community presence on different channels. We have identified at least seven community channels, which are Reddit, Facebook, Google+, Twitter, #IRC, Wiki page and the discussion board forum. On these community channels, Terasology provides support to the community, as well as a discussion platform for the features and project vision.

External Entities

Github provides the collaborative coding platform for Terasology. It helps developers to work on issues, and modders to create new modules, containing custom game mechanics and features, for the game.

Licenses

Terasology uses the Apache 2.0 and its own code of conduct licenses [4]. Based on the Apache license, the community has the freedom to distribute the software, to modify the software, and to distribute modified versions of the software under the terms of the license, without concern for royalties.

4. Development view

4.1 Module organization

To form a development viewpoint, we first modeled the system as a component diagram (see figure 3). This gives a depiction of how the components in the Terasology project are wired together to form the larger software system.

When looking at this diagram one can see that the game interacts, through an interface made available by the engine. In-game the user can define which modules should be loaded. The engine then asks the module loader for these modules by making use of an interface made available by the module loader. A module is a container for code and assets, which can be used by game types, mods or other higher-level concepts. Finally, the module loader makes the modules available to the engine, that passes them to the game. While being in-game, the engine communicates with the core to send the game component information to the game.

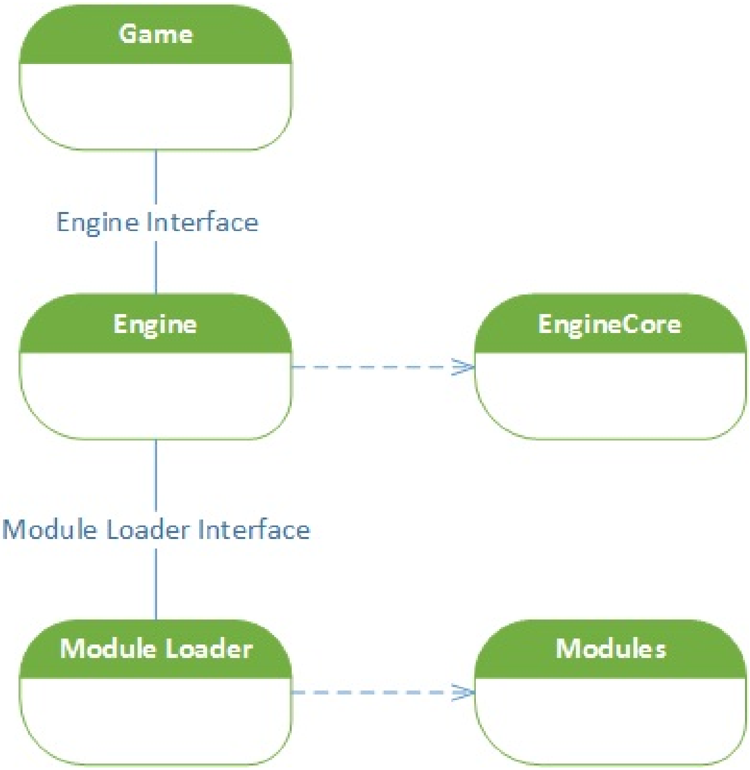


Figure 3: Terasology's component diagram

We modeled the organization of the system’s source code into a module structure model (see figure 4) [5]. This model defines the organization in terms of the architectural modules into which the individual source files are collected and the dependencies of these modules. The lines depict the dependencies between the packages, while the accompanying numbers depict the number of dependencies. We see the engine as the main element, where all the other architectural modules depend on. The other three main architectural modules are `engine-tests` , `facades` and `modules` . `Engine-tests` holds the JUnit tests for the engine. `Facades` contain executable front-end parts that can be chosen to be implemented. The `modules` are separate containers of content which can be injected in Terasology, which expand the gameplay beyond the standard-included core modules.

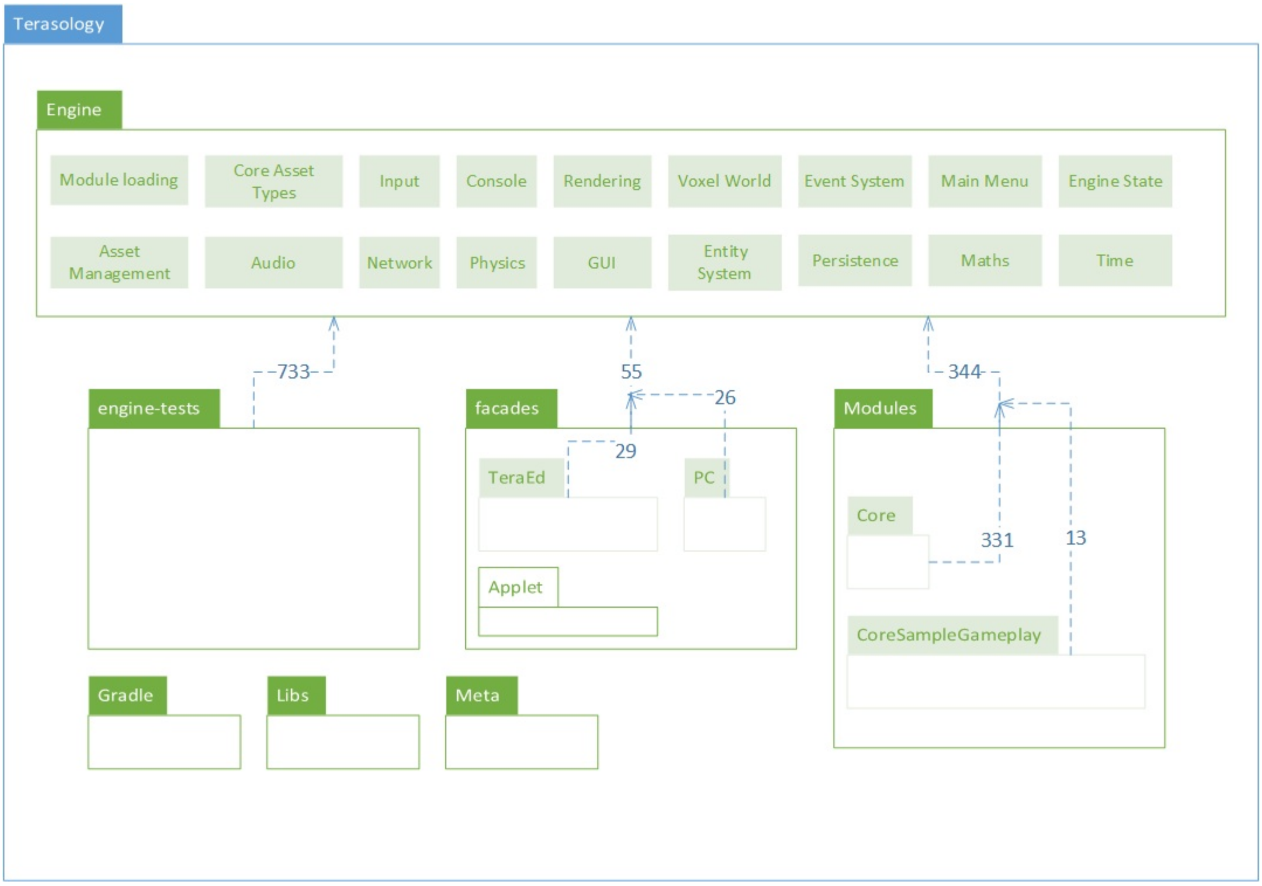


Figure 4: Terasology's dependency diagram

Standardization of design

To provide critical benefits to the maintainability, reliability, and technical cohesion of the system, it is required to implement a standardization of design [5]. Terasology achieves this by using three design patterns and off-the-shelf software elements.

First, the core framework of the game consists of an entity system. Everything in the game should be an entity; the only exceptions are the GUI and blocks. The entities are defined as interfaces and implemented through classes. If possible, code should be developed against these interfaces to make it possible to easily replace it in the future. An entity is a logical container for components. It can be identified through the Entity Manager, but the components are the elements that give meaning to it. Components are meaningful sets of data which should not contain any game logic, since they may only contain specific data types. The behaviour of the entities is managed by 'ComponentSystems'. The entity system, with entities, components and 'ComponentSystems' gives the architecture its flexibility.

Second, the design uses Events and Event Handlers instead of direct method calls, to allow systems to interact with each other. This again provides flexibility and extendability to the project, by decoupling the systems.

Third, blocks are the main elements in Terasology, and enable developers to design their own gameplay. They are put in `.block` files and are stored in a separate directory of a module. Blocks are JSON objects and are easily creatable and alterable.

Standardization of testing

Every major software project should implement standardization of testing to ensure a consistent approach to testing. Terasology has implemented his own standardized testing approach described below.

First, every contribution should be done through a Pull-Request (PR) from the Terasology GitHub repository. Terasology's GitHub bot "GooeyHub" will assist the Pull-Request by informing the admin and by providing an automated test request given by admin "Cervator" directly from PRs comment.

After the admin gives the command to test, GooeyHub will send the test request to Terasology's Jenkins, regarding the PR. Jenkins then builds and tests the PR code and provides the result, which GooeyHub retrieves and posts as a link on the PR's comment page.

After this phase, the admin and other related contributors will collaborate and discuss regarding the changes that are proposed by the contributor. They need to ensure that the new improvement complies with the architectural vision.

As a continuous integration and continuous delivery application, Terasology is using Jenkins in order to build and test every pull request. Jenkins implements a number of tests, such as: JUnit test, Checkstyle, FindBugs test, and PMD test.

Instrumentation

Terasology uses SLF4J (Simply Logging Framework For Java) library for different logging implementations. SLF4J also provides different log level configurations that are able to be adjusted to the requirements. Terasology's contributors have also created general guidelines about the log levels, which can be found on the Terasology forum [6].

4.2 Technical debt

Slowing rate of velocity

Velocity is a measurement of the amount of work a development team can complete during a time interval. When velocity starts to slow down over the course of several iterations, technical debt might be the reason [5]. If too much technical debt has piled up it will impair the development productivity. By analyzing the Github statistics we can see that the productivity comes with ups and downs. Normally this could be a sign of technical debt, but in this case that is very unlikely. This is because there is no structured development process with measurable iterations, and the activity and consistency of the team are subject to change. Any slowing rate of velocity in the Terasology project can therefore not be linked with certainty to technical debt.

Aging libraries

By taking a look at the versions of the libraries the project uses, we could determine if these had an impact on the technical debt, as this is increased by versions that are behind or no longer being maintained. We looked at the used libraries covered in appendix A. A total of six libraries has been found that where behind their latest stable version. No libraries have been found that were no longer being maintained. This gives a indication that the technical debt could be increased by the aging libraries.

Defects

The project has been analyzed with the use of the FindBugs plugin. The plugin gave a total of 301 warnings, from which were 91 high priority and 210 medium priority, which turns out to be three defects per thousand lines of non-commenting source statements. Table 2 gives a summary of the number of errors per warning type.

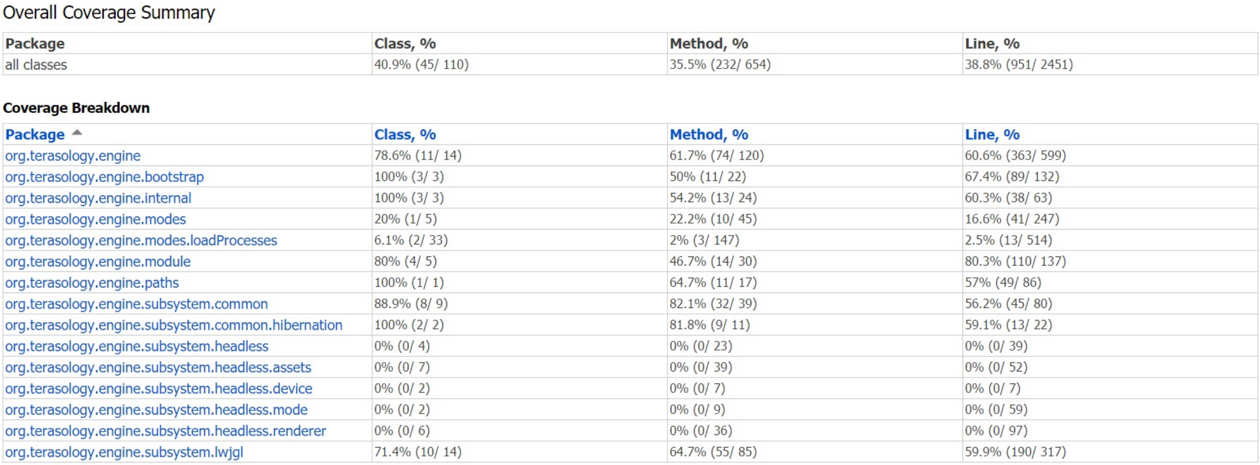
Warning Type	Number
Bad Practice Warnings	73
Correctness Warnings	3
Internationalization Warnings	2
Malicious Code Vulnerabilities Warnings	62
Multithreaded Correctness Warnings	1
Performance Warnings	52
Dodgy code Warnings	108

Table 2: Warnings Overview

Bugs should be fixed as they are found and should not be let to accumulate. The number of defects this project has, could be the cause of a small amount of technical debt.

Low automated test coverage

The project has been analyzed by the built-in test coverage tool of the IDE IntelliJ. The unit tests from Terasology only cover the engine package in a direct way. As is depicted in figure 5, only 40.9% of the classes in the engine is covered. The general agreement is that more than 90% coverage can be seen as a good sign, and less than 75% coverage may indicate a serious problem. In this case, it is very likely that the low automated test coverage is the cause of a great amount of technical debt, as tests are useful to be able to refactor code.



Overall Coverage Summary

Package	Class, %	Method, %	Line, %
all classes	40.9% (45/ 110)	35.5% (232/ 654)	38.8% (951/ 2451)

Coverage Breakdown

Package ▲	Class, %	Method, %	Line, %
org.terasology.engine	78.6% (11/ 14)	61.7% (74/ 120)	60.6% (363/ 599)
org.terasology.engine.bootstrap	100% (3/ 3)	50% (11/ 22)	67.4% (89/ 132)
org.terasology.engine.internal	100% (3/ 3)	54.2% (13/ 24)	60.3% (38/ 63)
org.terasology.engine.modes	20% (1/ 5)	22.2% (10/ 45)	16.6% (41/ 247)
org.terasology.engine.modes.loadProcesses	6.1% (2/ 33)	2% (3/ 147)	2.5% (13/ 514)
org.terasology.engine.module	80% (4/ 5)	46.7% (14/ 30)	80.3% (110/ 137)
org.terasology.engine.paths	100% (1/ 1)	64.7% (11/ 17)	57% (49/ 86)
org.terasology.engine.subsystem.common	88.9% (8/ 9)	82.1% (32/ 39)	56.2% (45/ 80)
org.terasology.engine.subsystem.common.hibernation	100% (2/ 2)	81.8% (9/ 11)	59.1% (13/ 22)
org.terasology.engine.subsystem.headless	0% (0/ 4)	0% (0/ 23)	0% (0/ 39)
org.terasology.engine.subsystem.headless.assets	0% (0/ 7)	0% (0/ 39)	0% (0/ 52)
org.terasology.engine.subsystem.headless.device	0% (0/ 2)	0% (0/ 7)	0% (0/ 7)
org.terasology.engine.subsystem.headless.mode	0% (0/ 2)	0% (0/ 9)	0% (0/ 59)
org.terasology.engine.subsystem.headless.renderer	0% (0/ 6)	0% (0/ 36)	0% (0/ 97)
org.terasology.engine.subsystem.lwjgl	71.4% (10/ 14)	64.7% (55/ 85)	59.9% (190/ 317)

Figure 5: Test coverage of the Terasology project

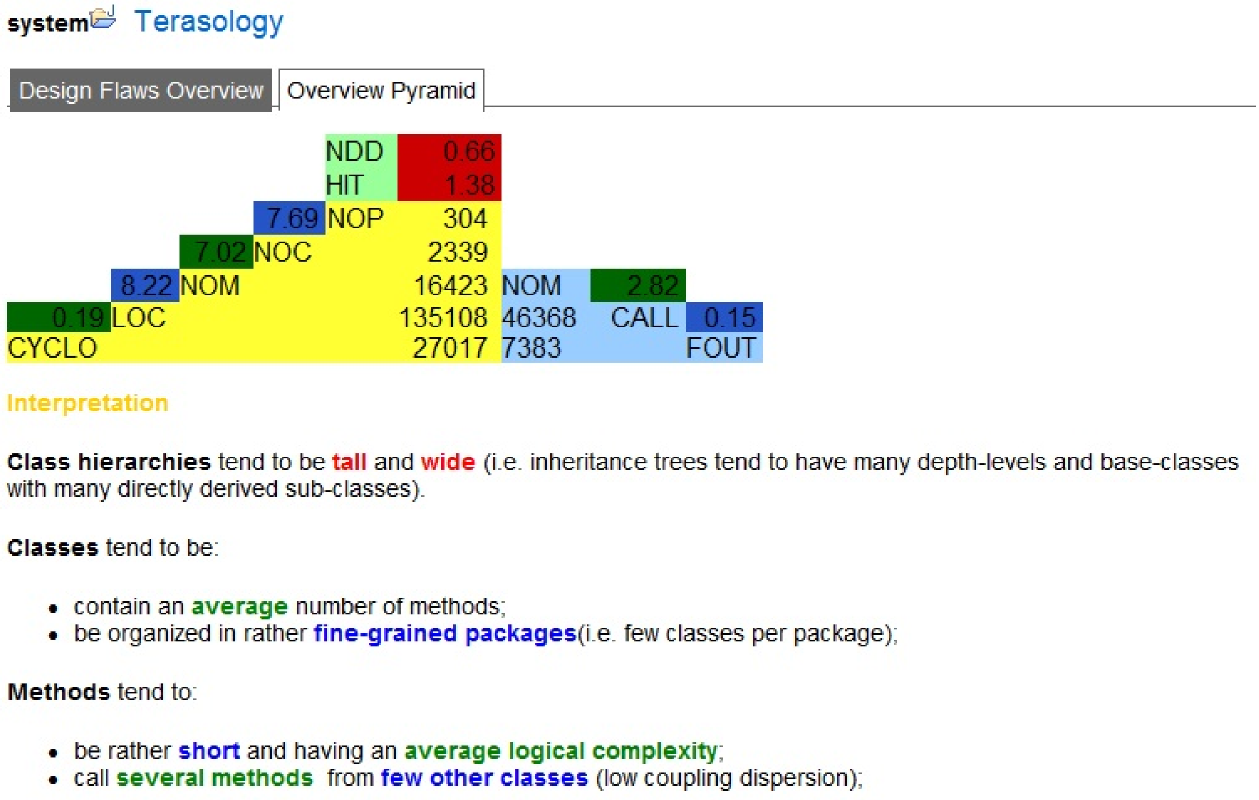
Poor score on quality metrics

The project has been analyzed with the tool InCode to get an overview of the other quality metrics. From the 2339 classes that were analyzed, 154 had design flaws, and from the 16423 methods that were analyzed, 235 had design flaws. These metrics can be found in appendix B.

From this analysis we can infer that 6.6% of the classes and 1.4% of the methods have design flaws. The five packages with the highest severity of design flaws are:

- `/engine/src/main/java/org.terasology.protobuf` , cumulative severity of 518.
- `/engine/src/main/java/org.terasology.rendering.nui.internal` , cumulative severity of 107.
- `/modules/Core/src/main/java/org.terasology.logic.inventory` , cumulative severity of 65.
- `/engine/src/main/java/org.terasology.logic.characters` , cumulative severity of 26.
- `/engine/src/main/java/org.terasology.engine.subsystem.headless.renderer` , cumulative severity of 24.

With the help of InCode we have also created an overview pyramid (see figure 6). Each of the rows has a colored percentage, which is derived from the ratio of the number on this row and the one under it, e.g., CYCLO / Line is 0.19. The given numbers indicate where the ratios fit into the industry-standard ranges. From figure 6, can be derived that the number of direct descendants and height of inheritance tree is outside the range. The number of classes per package, lines of code per method and fan out (number of other methods called by a given method) per call are all below the range. The other ratios are all within the range. The red and blue cases increase the technical debt and should therefore be refactored.



system Terasology

```
                        NDD    0.66
                        HIT    1.38
                   7.69 NOP     304
              7.02 NOC         2339
         8.22 NOM             16423 NOM      2.82
    0.19 LOC                 135108 46368  CALL  0.15
CYCLO                         27017 7383         FOUT
```

Interpretation

Class hierarchies tend to be **tall** and **wide** (i.e. inheritance trees tend to have many depth-levels and base-classes with many directly derived sub-classes).

Classes tend to be:

- contain an **average** number of methods;
- be organized in rather **fine-grained packages**(i.e. few classes per package);

Methods tend to:

- be rather **short** and having an **average logical complexity**;
- call **several methods** from **few other classes** (low coupling dispersion);

Figure 6: Pyramid generated with InCode

By using the tool CodePro Analytix we analyzed the comments ratio, which was 6.9%, such low amount of Javadoc could contribute to extra technical debt. As Terasology is an open source project which is dependent on new developers, documentation is an essential part of the understandability of the code for them.

Dealing with technical debt

To get a good overview of how the Terasology team deals with technical debt, we analyzed the Github pull requests and issues. Since the Terasology project does not contain any labels concerning technical debt, we had to filter the issues and pull requests based on our self-created set of keywords. The keywords we filtered on are: "technical debt", "clean", "maintenance", "improve", "style", "cleanup" and "refactor". From these results we tried to find instances where the team had to deal with technical debt, which caused them to have discussions and possible refactoring decisions they had to make.

We found out that there have been almost no relevant issues regarding refactoring. About 2% of the open and closed issues is about refactoring code. Issue #1587 shows an example for an open issue about refactoring the "Breathing" modules, and issue #2089 shows an example for a closed issue about improving the readability of the "Assets". The low number of issues regarding refactoring could mean that the team is not aware of the design flaws in the code or is simply ignorant towards them. When looking at the pull requests regarding refactoring, we found out that about 2% of the pull requests is about refactorization that is not based on issues. These refactorization pull requests were all own-initiative refactorization contributes. Most of the times contributors focus on refactoring classes, e.g., #1562. Sometimes contributors are working on certain issues and find code that needs refactoring, e.g., #1751. Other times contributors refactor closed issues of their own, which even can be work from six months ago, e.g., #2135.

Based on the data from the Terasology project on Github we can state that the developers do not seem to discuss any problems regarding technical debt. Any mentioning about code refactoring is confined to its own issue or pull request, and thus is not part of encountered technical debt during the work on issues. Though in the previous section we stated that there is a large amount of possible technical debt in the Terasology project. The question then of course is, why this debt is not mentioned anywhere on Github. The reason for this could be that any bad code is refactored on the fly, without mentioning it on Github. Another unlikely but possible reason could be that the currently worked on issues do not have much overlap with the code that contributes to the technical debt, and that the technical debt still has to be repaid in the future. The last and most likely reason is the fact that they take the current technical debt for granted. The developers could take the debt for granted as they are reluctant to refactor the code. The first reason for their reluctance could be due to the lack of tests that are available, as tests are needed to refactor. The second reason could be the lack of understanding of the code. A lot of the initial developers have left the Terasology project, which could be a reason for the current developers to leave the old code as it is (if it ain't broke, don't fix it). The last reason could be the short term focus of the developers as new features are more important than refactoring existing ones. This becomes more likely if we look at the continuous integration tool Jenkins that they use. By analyzing this tool we can clearly see that design flaws are monitored, yet the developers run Jenkins and manually accept the result even though Jenkins shows that a lot is wrong with the code.

In a reaction on our questionnaire on the Terasology forum "Skaldarnar", confirmed that they ignore technical debt until the moment the code is touched for other reasons [7].

5. Variability perspective

Terasology has a range of important features that could cause the user experience to change. The features and their relationships have been modeled in a feature model as depicted in figure 7 [5]. The mandatory features are present in all products (be it as default settings). In OR-features, it is possible to select one or more feature, wherein the alternative (XOR)-features only one of the features can be selected.

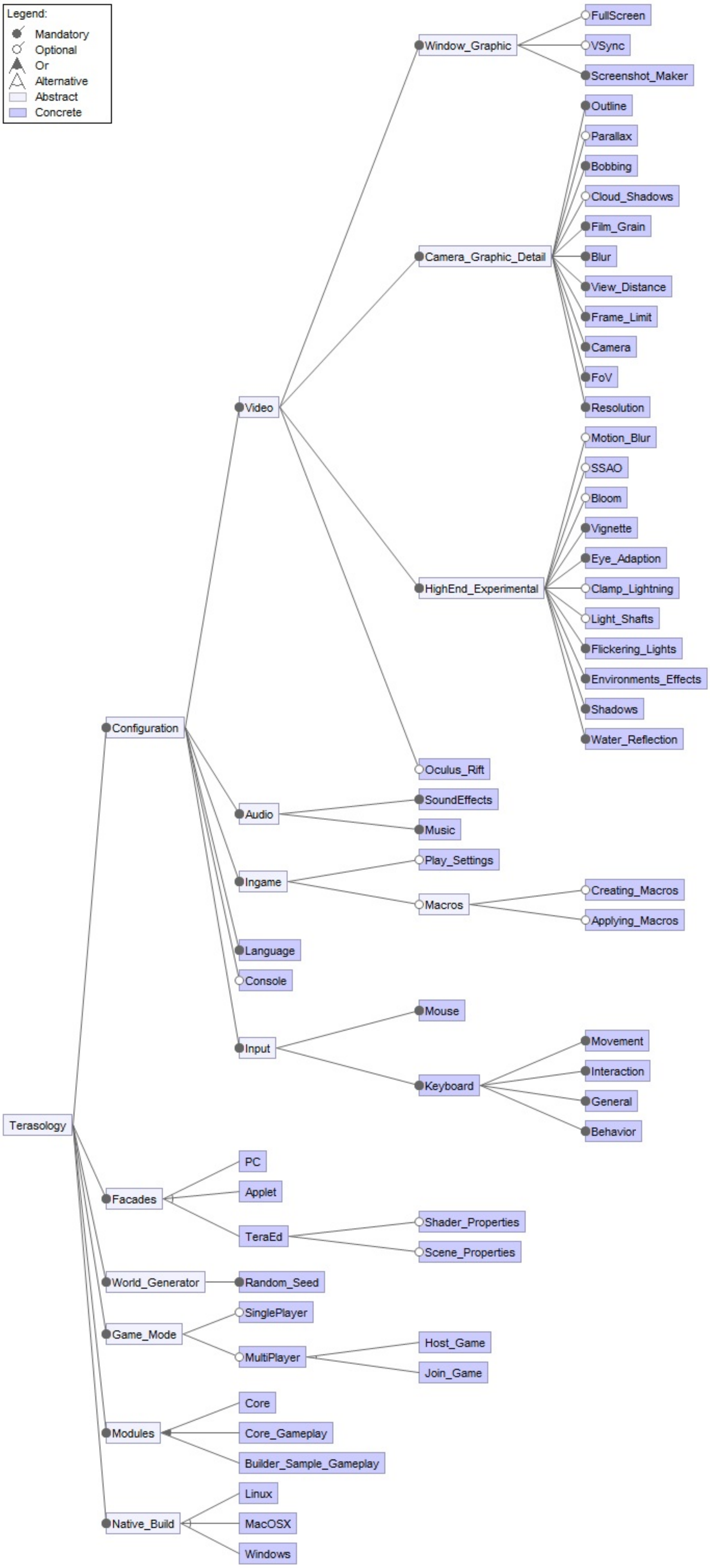

Legend:
Mandatory
Optional
Or
Alternative
Abstract
Concrete
Terasology
Configuration
Video
Window_Graphic
FullScreen
VSync
Screenshot_Maker
Camera_Graphic_Detail
Outline
Parallax
Bobbing
Cloud_Shadows
Film_Grain
Blur
View_Distance
Frame_Limit
Camera
FoV
Resolution
HighEnd_Experimental
Motion_Blur
SSAO
Bloom
Vignette
Eye_Adaption
Clamp_Lightning
Light_Shafts
Flickering_Lights
Environments_Effects
Shadows
Water_Reflection
Oculus_Rift
Audio
SoundEffects
Music
Ingame
Play_Settings
Macros
Creating_Macros
Applying_Macros
Language
Console
Input
Mouse
Keyboard
Movement
Interaction
General
Behavior
Facades
PC
Applet
TeraEd
Shader_Properties
Scene_Properties
World_Generator
Random_Seed
Game_Mode
SinglePlayer
MultiPlayer
Host_Game
Join_Game
Modules
Core
Core_Gameplay
Builder_Sample_Gameplay
Native_Build
Linux
MacOSX
Windows

Figure 7: FeatureIDE diagram

The main dependencies of Terasology are the native build related features. In order to run seamlessly on the different platforms (Linux, MacOSX and Windows), Terasology is required to define different native build configurations that include Input Library (`libjinput-linux` , `libjinput-osx` , `jinput`), Graphic Library (`liblwgjl`) and Audio Library (`OpenAL`). Moreover, there is also some specific platform 32-bit/64-bit configuration for each library being used on specific native builds, as depicted in figure 8.

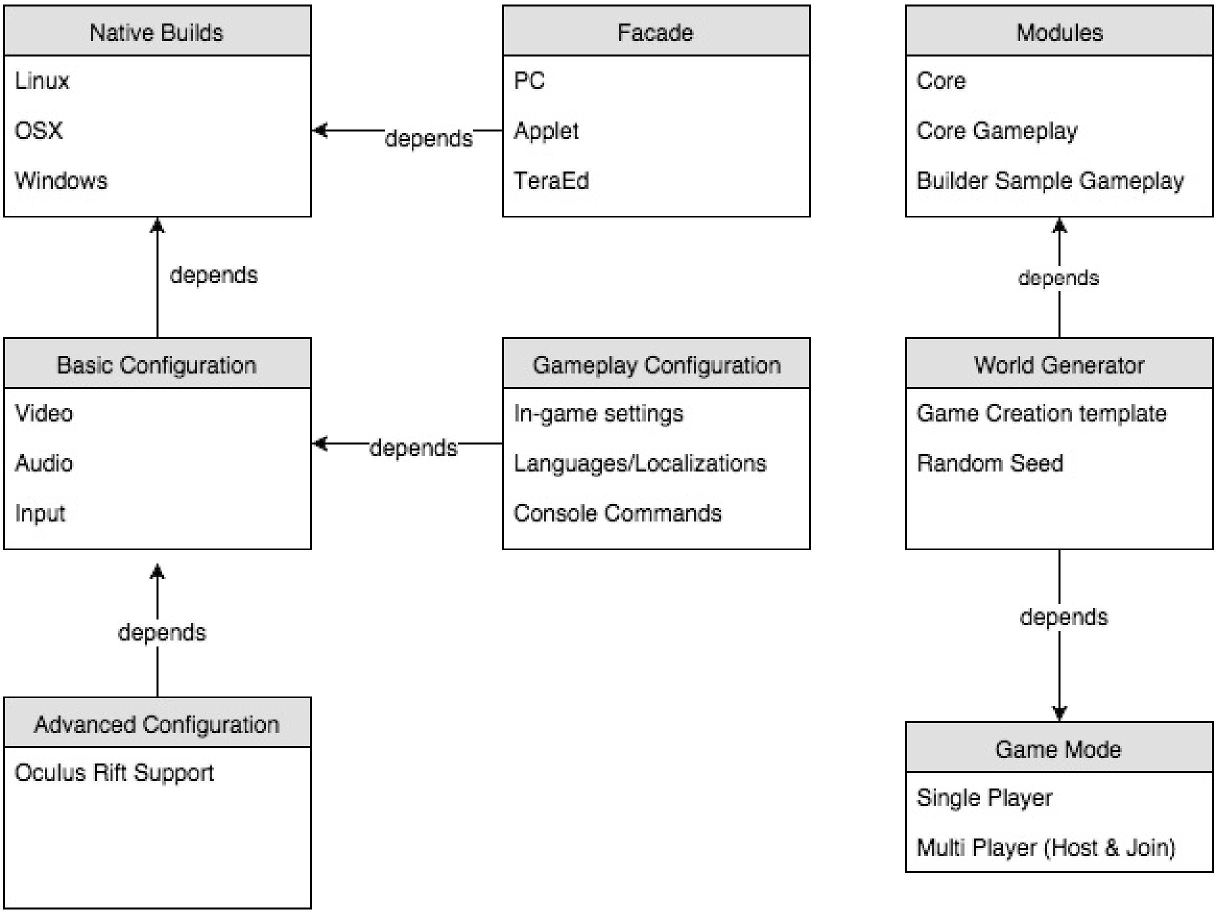


Figure 8: Feature Dependency UML diagram

Each Native build has two groups of features that depend on it, which are `Basic Configurations` and `Facades` . These features basically require specific platform libraries to communicate with the hardware (Input, Audio and Graphics). Advanced configuration feature support, such as Oculus Rift, extend the capabilities of the Video feature of the basic configuration. Gameplay related configuration also depends on basic configuration features, these include keyboard localization, language of the interface and other user interface related features. Separate from the Basic and Gameplay Configuration there is a group of features that do not depend on the native build, which are Modules and Game Mode. These two groups are mainly dependent on world generator features.

5.1 Variability Strategy

Terasology makes use of configuration files which can be edited in the various setting screens. These configuration files hold the various global configuration information that the user can modify. The configuration files can be saved and loaded in a JSON format with the `.ui` extension, and define the menu layout. Configurable items, like the button handlers, are defined in the `rendering.nui` package. These classes can adjust the overall config classes defined in the `Config` package.

Modules can be seen as containers for code and assets, that can be used by game types, mods or other higher-level concepts. Modules can be enabled, disabled or replaced. Therefore some of the main parts of the system, e.g. `Core Gameplay`, are also modules. Every module gets a copy of the build file from the Core module so that all modules are built equally. Modules use a simple `module.txt` file to define their configurations and dependencies towards other modules. Gradle, a build automation system, handles the possible problems regarding the dependencies [2].

Terasology uses facades to create a front-end which can be executed. There are multiple facades, with the main one being `PC Facade`. This facade is included with the base version of Terasology, and can be used to run the game as an application. Other facades are `FacadeApplet`, `DestSolAndroid`, `DestSolGwt` and `FacadeAWT`.

Besides the single player mode, there is also a multiplayer mode option. Both options are part of the build and are available before creating a game. There is a small network infrastructure provided that enables someone to act as host and others to join his game. Multiplayer mode is still a feature that is under development and is not seen as stable by the community.

6. Deployment view

This deployment view section looks beyond the alpha release. The Terasology launcher can be downloaded from the download page. Extracting this launcher provides the opportunity to install stable releases of the game, with the latest version by default. Like the game, the launcher is executable on Windows, Linux and MacOSX. The only other software component, besides the launcher and the release version, that needs to be present on the PC is Java Virtual Machine version 8. There are other Java libraries used by Terasology, but they are downloaded as part of the stable release. The basic version comes with two modules, `Core` and `CoreSampleGameplay`. Within the game, it is possible to download additional modules. Additional modules change or add features of the game. Downloaded modules can be selected before creating a game.

There are no predefined hardware requirements for Terasology. The game, as we experienced by testing, has high graphical requirements. We asked the community for advice on running the game on basic systems with low graphical power: "With just the Core Gameplay and low graphic settings (especially view distance) any system that decently supports OpenGL 2.1 and Java 8 should have a chance to work" [7]. However, there are modules, such as Throughout the Ages in combination with high graphic settings that can crash any regular system. For the multi-player mode, the server demands are not that problematic, however a noticeable draw on CPU and memory remains. Additionally, the game tends to crash when adding more than a couple of players to the game. Optimization of the server is one of the key parts the community is focussing on.

7. Scalability Perspective

In order to identify the scalability of Terasology and understand its capability to perform under an increased workload, we observed the project scalability with various dimension measurements [5].

7.1 Administrative Scalability

In general, everyone is able to contribute to, comment on and provide ideas for this project. However, there is a key development phase that is only handled by a specific person, consisting of the contribution approval (pull-request) and the maintaining of the architectural vision. Unfortunately, there are only a few active maintainers that are available for this important responsibility. As a reminder, popularity on Github is essential for this project; because it attracts more contributors and accelerates the project development. Based on this fact, we have foreseen that this administrative problem will become the main scalability bottleneck for the Terasology project.

Administrative Scalability really became a bottleneck when the project recently entered Google Summer of Code 2016, where the popularity largely outnumbered the capability of the community to support more than a couple of student.

7.2 Functional Scalability

Terasology's modular architecture enables contributors to develop new modules as desired. New modules are integrated with existing Terasology modules to provide new features with minimal effort. This capability strengthens the ability to stimulate the open-source community to keep developing Terasology.

7.3 Geographic Scalability

Terasology has three factors that improve geographic scalability. First, it is an open source project that gets contributions from every continent. Second, every user is able to host their own Terasology game, which acts as a server in the multiplayer mode. This enables scaling without the boundary of geographic constraint. Third, the game provides different languages for their users, to also attract non-English speaking players.

7.4 Load Scalability

The multiplayer game mode, needs to be able to easily expand to accommodate heavier loads of inputs. Therefore load scalability is the main requirement to provide multiplayer experience on Terasology. Every user is able to host their own game using their own resources (computer/server) and invite other users to join the game. Other users are only required to add the server connection information in order to connect to the host server. With this distributed architecture, Terasology is easily expandable to meet the community needs.

7.5 Generation Scalability

Terasology has a modular architecture. This makes it possible to change or add modules to the project. It can handle module dependencies and has a version control mechanism, to prevent the game from breaking when new module versions are introduced. This makes it possible to handle different generations of different modules. However, changing modules and checking for newer versions has to be done manually. This makes the execution of module generation coordination very labour intensive.

8. Contributions

The list of contributions that have been made to the Terasology project can be found in appendix C. We have five merged and one closed pull requests, from which four solved user interface issues and one solved a log file issue.

9. Conclusion and Reflection

Terasology is an open source game, which has grown from a research project to a mature project growing into alpha testing status. It has a small but active group of contributors, who work in a completely automated development environment on Github. The architecture has a modular structure, with a small engine and a lot of modules that add or change gameplay. This modular architecture has made it possible to reach a high variability, enabling to change almost every feature in the game. As a result of the high variability, some combinations of high graphics options, as well as the combination of multiple demanding modules, make the game impossible to run on a standard PC. Therefore a big architectural change is being discussed by the architects in order to decrease the computational demand of the game and improve its stability. Besides the possible architectural overhaul our research has indicated that the technical debt of the project is too high above the desired standard. The project contains a high number of design flaws and low amount of test cases and JavaDoc. This could prove to become problematic in the future when new contributors will start working on the project.

During the DESOSA 2016 project, Terasology has been accepted for Google Summer of Code. This will greatly increase the number of new contributors and users. Therefore GSoC would be a good research subject to analyze how Terasology is going to deal with new contributors and users, in regard to the technical debt and deployment and scalability factors.

All in all we have had a lot of fun working on the project, but it would have been a lot less fun if the Terasology community had not given us a very warm welcome like they did. They did a perfect job in assisting us on our contributions and answering our questions. Therefore we would like to thank the Terasology community for helping us.

10. References

[1] Terasology.https://github.com/MovingBlocks/Terasology. [Online; accessed 29-March-2015].

[2] Terasology’s Wiki.https://github.com/MovingBlocks/Terasology/wiki. [Online; accessed 29-March-2015].

[3] Terasology’s Github Pulse. https://github.com/MovingBlocks/Terasology/pulse. [Online; accessed 29-March-2015].

[4] Terasology’s Code of Conduct. https://github.com/MovingBlocks/Terasology/blob/develop/docs/Conduct.md. [Online; accessed 29-March-2015].

[5] Rozanski, N., Woods, E. 2012. Software System Architecture.

[6] Terasology’s logging guidelines. http://forum.terasology.org/threads/logging.611. [Online; accessed 29-March-2015].

[7] Terasology’s forum post: Technical requirements. http://forum.terasology.org/threads/technical-requirements.1476/. [Online; accessed 29-March-2015].

Appendix A - Used libraries

Several libraries are being used to run the game, the main ones being:

- LWJGL, which is the Lightweight Java Game Library. Current version being used: 2.9.3, while the latest version is 3.0.
- JUnit, a library for unit testing. Current version being used: 4.12, which is the latest version.

Other libraries:

- Storage and networking:
    - guava, version: 19.0, which is the latest version.
    - gson, version: 2.5, while the latest version is 2.6.1.
    - protobuf-java, version: 2.6.1, which is the latest version.
    - trove4j, version: 3.0.3, which is the latest version.
    - netty, version: 3.10.5, while the latest version is 4.0.34
- Java-related libraries:
    - jna-platform, version: 4.2.1, which is the latest version.
    - reflections, version: 0.9.10, which is the latest version.
    - javassist, version: 3.20.0-GA, which is the latest version.
    - reflectasm, version: 1.11.0, while the latest version is 1.11.1
- Graphics, 3D, UI:
    - lwjgl_util, version: 2.9.3, which is the latest version.
    - java3d, version: 1.3.1, while the latest is 1.5.2
    - abego.treelayout.core, version: 1.0.3, which is the latest version.
    - miglayout-core, version: 5.0, which is the latest version.
    - PNGDecoder, version: 1111, which is the latest version.
- Logging and audio:
    - slf4j-api, version: 1.7.13, while latest version is 1.7.17
    - jorbis, version: 0.0.17, which is the latest version.

Appendix B - Quality metrics

From the 2339 classes that were analyzed, 154 had design flaws:

- 87 Data Classes: These classes are exposing a significant amount of data in its public interfaces. This impacts the encapsulation, coupling, and cohesion of the system.
- 34 Schizophrenic Classes: These classes have public interfaces that are large and are used non-cohesively by client methods. This impacts the complexity, encapsulation, coupling, inheritance and cohesion of the system.

- 19 God Classes: These classes use many attributes from external classes, directly or via accessor methods. They are also excessively large and complex, due to the fact that their methods have a high cyclomatic complexity and nesting level. This impacts the complexity, encapsulation, coupling and cohesion of the system.
- 14 Tradition Breaker Classes: These classes hide parts of the inherited interface. This impacts the coupling, inheritance and cohesion. From the 16423 methods that were analyzed, 235 had design flaws.
- 118 Data Clumps: These methods have long parameter lists, which could be a sign that the group of parameters could form a new abstraction, that could be extracted to a new class. This impacts the complexity and encapsulation of the system.
- 2 Sibling Duplications: These methods have duplications with methods from the inheritance hierarchy. This impacts the complexity, encapsulation, coupling and inheritance of the system.
- 26 Feature Envies: These methods heavily use attributes from one or more external classes, directly or via accessor operations. This impacts the complexity, encapsulation, coupling and cohesion of the system.
- 2 Message Chains: These methods use one object to access another object, which on his turn uses the obtained object to access another object, and so on. This impacts the complexity, encapsulation and coupling of the system.
- 34 Internal Duplications: These methods have duplication with other methods from their class. This impacts the complexity of the system.
- 53 External Duplications: These methods have duplication with other methods external to their class and their hierarchy. This impacts the complexity, encapsulation and coupling of the system.

Appendix C - Contributions

Pull request #2176 for issues #844 and #2134. MERGED

This pull request was made to fix issues #844 and #2134. In these issues, a request was made for a solution for dropdown menus which were becoming too long. To solve this, we implemented a new type of menu called UIDropdownScrollable (based on UIDropdown). This menu was similar to the typical dropdown menu, but had a scrollbar on the right side to shorten the length of the menu.

One problem remained however in this pull request, which was that the menu did not close whenever the scrollbar was focussed last. The pull request message contained a detailed description of how to replicate the issue to clarify the issue.

Pull request #2180 for issue #2179. CLOSED

This pull request fixed an issue we found in our previous pull request (#2176). The scrollbar showed a scrollbar even when there was nothing to scroll down for. The issue was fixed by adding checks if there were more items than that the menu should be long.

This fix caused some more code to be added, and the decision was made to clean up the code, improving maintainability. More methods were added to divide the massive onDraw functions into smaller pieces.

Afterwards, we realized something went wrong with the commits, as it appeared that a rebase went wrong. This eventually got fixed in pull request #2183.

Pull request #2181 for issue #2178. MERGED

Issue #2178 was created by Cervator who directly asked us to implement our newly created scrollable menu from pull request #2176 in places which we deemed necessary. We implemented the scrollable menu is several places, and hinted in the pull request message that pull request #2180 was probably useful for this pull request, since it fixed the bad scrollbar.

Pull request #2182 for issue #2173. MERGED

This pull request added a difference between the subheader titles of the singleplayer and multiplayer menus. Initially, an extra class and an extra .ui file were created, but msteiger asked us to change this to prevent code duplication. We complied by adding another commit, which removed the Java class file which was deemed to be dead code. Afterwards, msteiger thanked us but asked us to also get rid of the extra .ui file, suggesting another way of solving this issue. After a small discussion, msteiger advised us to use a placeholder text for the subtitle label and set the correct one through Java code when the menu screen is opened. We complied by adding several other commits, most important being commit 1 and commit 2. Cervator merged the pull request, but mentioned that the subtitles should also be added to other game screens. For this, he created a new issue #2233, which we will fix in the nearby future.

Pull request #2183 for PR #2180. MERGED

Pull request #2180 solved its assigned issue, but something went wrong with the commits. This was caused by rebasing the develop branch on our fork, and updating the worked-on branch. After this was figured and pointed out, this pull request got closed.

Pull request #2183 was created to fix the same issue as #2180, but this time with a proper commit history.

Pull request #2184 for issue #2030. MERGED

Issue #2030 explained an issue regarding the filename of the logfile in multiplayer mode, as it was deemed to be unclear. We created a new structured filename, describing the server name, address and port to give a clearer view of what the log message contains.

msteiger pointed out a different way of handling this issue regarding Protobuf's `serverInfoMessageBuilder.setGameName()` method.

Cervator merged the pull request, and pointed out that the filename could be changed in a follow-up issue/pull request if a better name is found.

WildFly

by Shishir Subramanyam,Ioana Leontiuc, Mengmeng Ye, Kyriakos Fragkeskos *Delft University of Technology*

Abstract

WildFly is a server application that provides easy communication between the user and its server. It is one of the many features of the JBoss Application Enterprise. WildFly provides remote management for one or more servers and runs on a local host server as well. This includes domain mode, where a domain of servers on different hosts can be managed by a central domain controller. We analyzed the open source modules project of WildFly. Based on the online available information and the source code (both manual and tool based analysis) we reversed engineered the context, development, functional and operational viewpoints and the evolution perspective.

Table of Contents

Context

WildFly is an open source application server in Java [2] . Its copyright is owned by Red Hat Inc. This company has a pattern of purchasing projects, releasing as open source one of the system`s functionality and then charging clients for support and access to the full feature version.

RedHat Inc has the infrastructure to turn rather small scale popular projects into a fully functional product. As stated by them: “We help more than 90% of Fortune 500 companies solve business challenges, align their IT and business strategies, and prepare for the future of technology”. As much as possible RedHat Inc tries to build up their new projects on their other owned projects. This is feasible since they are “the world`s leading provider of open source solutions”[1].

In 2006 RedHat Inc bought JBoss, and owned the copyright for the JBoss Middleware, a portfolio of enterprise-class application and integration middleware software products. [5]. One of these is The Enterprise Application Platform a single platform to quickly develop and deploy applications [4]. The application server within the JBoss Enterprise is called WildFly and released as open source.

Stakeholders

RedHat Inc has full time employed developers that work on extending their project. However, since they are open source, some developers come from the open source community. We refer to the people that voluntarily contribute to the WildFly project by submitting pull requests or providing user support by answering related questions on forums or other platforms (YouTube videos, blog posts) as open source developers. We also consider the users of WildFly, and not the entire JBoss platform to be open source users. Therefore we refer to the open source community (OSCommunity) as being formed of these two groups, the open source developers and the open source users. We acknowledge the different roles each stakeholder group has. However our intention in the merging of some groups is to show that the same group of people has multiple responsibilities. We provide names of group representatives and acknowledge the contribution of the OSCommunity in the corresponding stakeholder groups.

Communicators & Developers: we merged these categories because both consist of the JBoss developers employed by RedHat Inc, represented by Jason Greene, the project leader and the OSCommunity. However any contribution made by the OSCommunity has to be approved by a JBoss developer. The primary communicator are JBoss employees, they approve the official documentation (Stuart Douglas, Brian Stansberry,Leos Literak to name a few [16]). We have found that there is a RedHat Certificate a lot of people pride themselves with on Linked In. Some of them even made instructional videos on YouTube[15]. Thus becoming communicators.

Maintainers & Production Engineers(& System Administrators): the people responsible for all these tasks are the JBoss developers employed by RedHat Inc. (like Stuart Douglas the JBoss software engineer that announced the new release). In this case the system administrators are the production engineers.

Suppliers: or sponsors are first of all RedHat Inc. The development made by their employees is the basis of new WildFly releases (latest version WildFly 10 on 29 January 2016). Also RedHat Inc provided other projects to help the JBoss development process which includes the development of WildFly. Some of these projects are:

- Hibernate = domain model persistence for relational databases [7]
- Narayana = transaction manager [8]
- Infinispan = distributed in-memory key/value data store with optional schema [9]
- Arquillian = component model for integration tests that execute inside the real runtime environment [3]

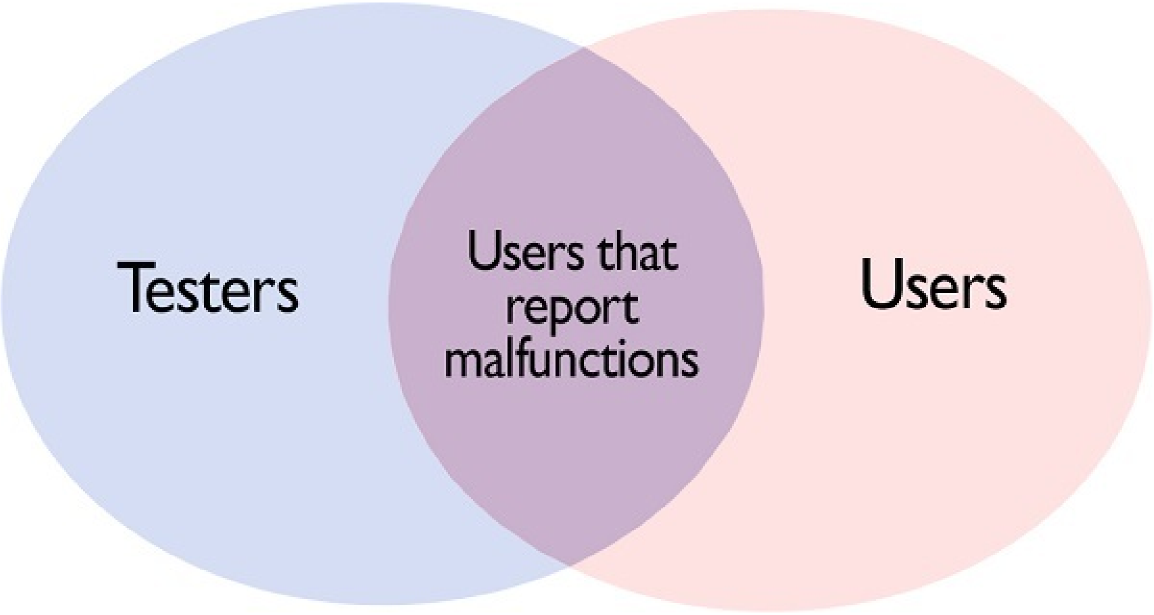
Beside the RedHat products WildFly also benefits from[17]:

- Awestruct = framework for creating static HTML [18]

- inteliJIDEA
- ej-technologies = free JProfiler licenses

Support Staff: the support available for free is from the OSCommunity. However, there is payed assistance available from JBoss. Because the second one is charged we consider them a proxy. They can be considered as an alternative if the OSCommunity does not rise to the challenge.

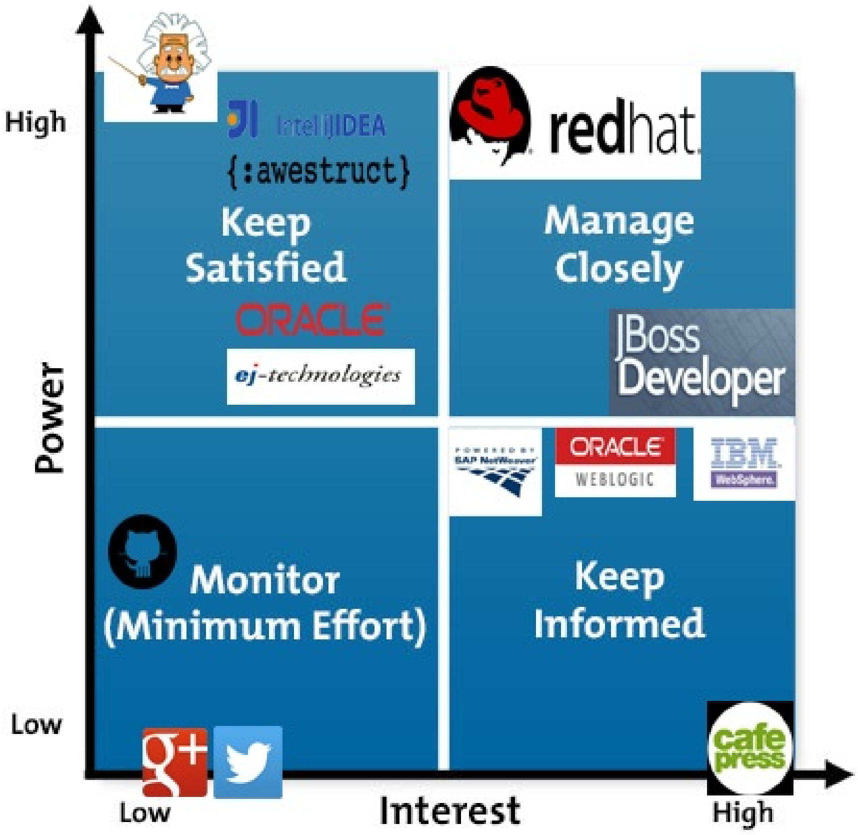
Testers & Users: the testers of JBoss (that use the Aquilian platform) and the users that pay for the JBoss Platform are implicitly in the same relationship with the WildFly project. Having said that, we have testers and users exclusively for the WildFly part. They come from the OSCommunity. However there are people outside this community that need the feature WildFly offers, thus becoming exclusive users of WildFly. We consider these categories to be in the following relationship:



Proposed Stakeholders

Analyzing the environment of WildFly we feel that in order to have a comprehensive stakeholder analysis there is a need for the following categories:

Followers: WildFly uses social media. These followers are exclusively interested in WildFly, 3960 twitter followers and 1352 members of the Google+ community. RedHat Inc has a total of 8300 employees [10]. As shown in the Power/Interest Grid for Stakeholder Prioritization this group has no power and low interest. However we consider that this group should not be monitored with minimum effort. This group is composed of employees of JBoss, people from the OSCommunity but also other people that are somehow interested in WildFly, they might not even be users (this group is represented in the Context View as a question mark). But they are potential users. These people are most likely to use WildFly. More importantly this group has the potential of becoming future JBoss employees. From the company`s perspective, this group is the first place to look for new talent because they build up an interest in the product on their own.

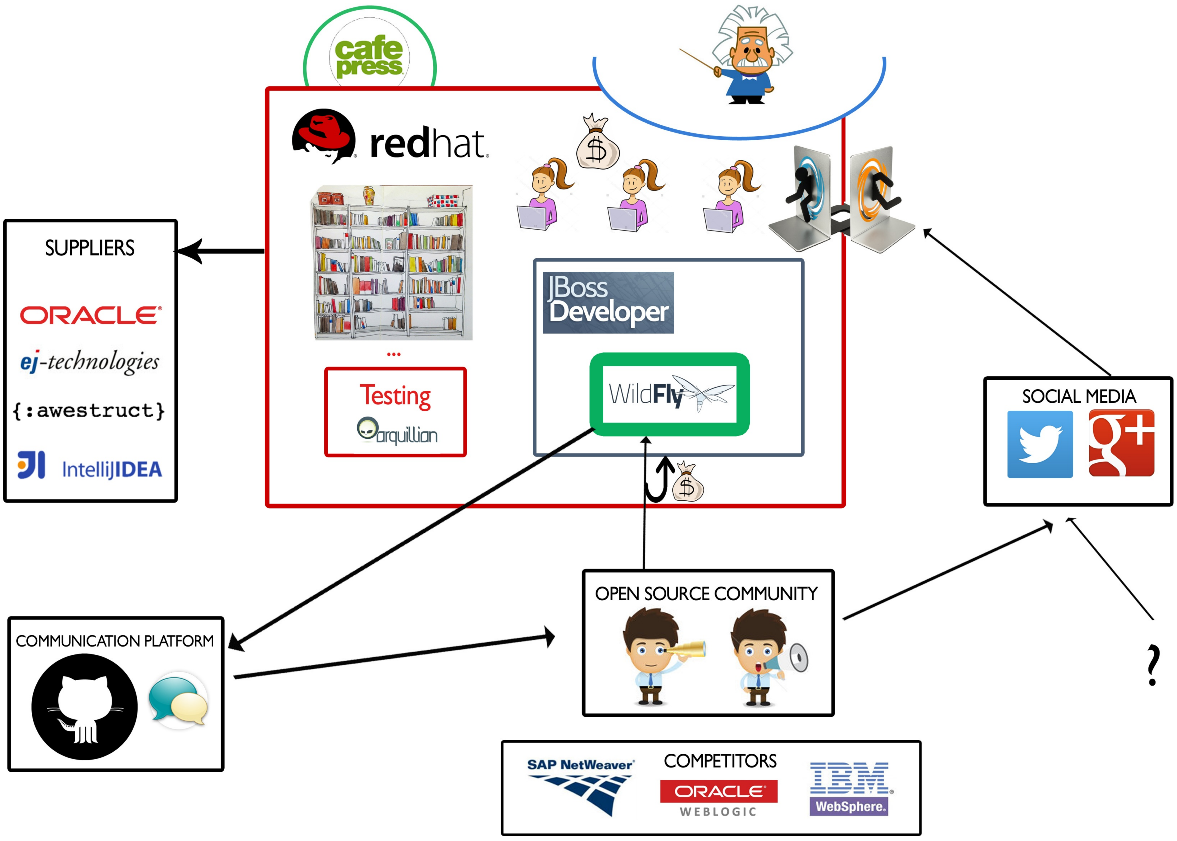


Competitors: the companies that offer the same services: IBM WebSphere, Oracle WebLogic,SAP NetWeaver

Senior Adviser: are the people with a lot of experience in the field that can have valuable input for the success of the development process. However these people are not exclusively involved with one project. They are not always present but when they are their input is always taken into account. They are low interest but at the same time a lot of power. For WildFly we found Arun Gupta. He is the vice president of developer advocacy at Couchbase. He has been building developer communities for 10+ years at Sun, Oracle, and Red Hat [11]. Even though he work for a different company he is interested in the project. He wrote a blog post on how to use WildFly on raspberry pie [12] and made a YouTube tutorial [13]

Free riders: This type of stakeholders are independent of the development process. They have no power but high interest. In a way they behave like a parasite. They have only to gain from the success of the project. In the case of WildFly we found Caffe Press, they sell JBoss Community Gear, cups and t-shirts with the logo: "wear your open source pride" [14]

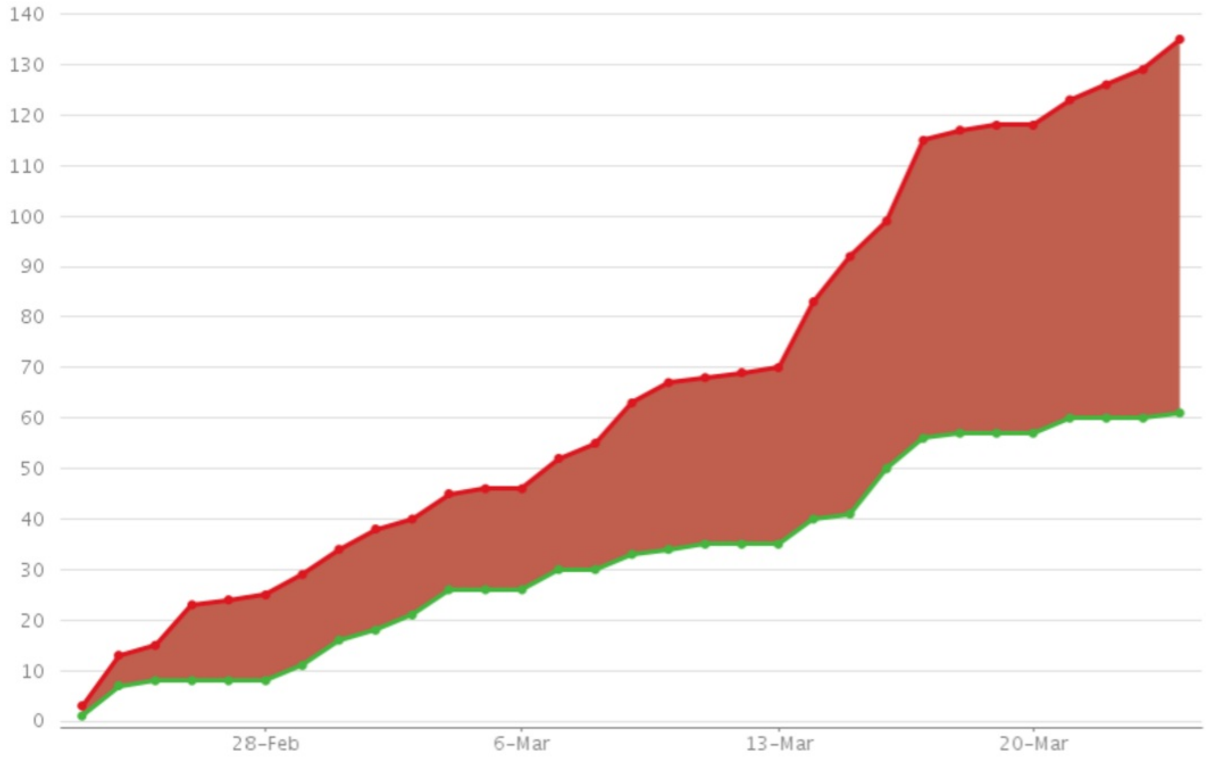
All these are summarized within the following context view:



Git Repository Analysis

Description	Result
Open pull requests	85
Closed pull requests	8573
Merged pull request ratio	1.07 per day
Proposed pull request ratio	1.5 per day
Contributors	248

Based on our personal experience the process of contributing is very thorough. The detailed oriented process for open source contributors makes the whole process time consuming and requires a significant number of re-submissions. However, it is very easy to report an issue. This situation can be observed in the correlation between created and resolved issues for the past 30 days (as of 24th of March 2016).

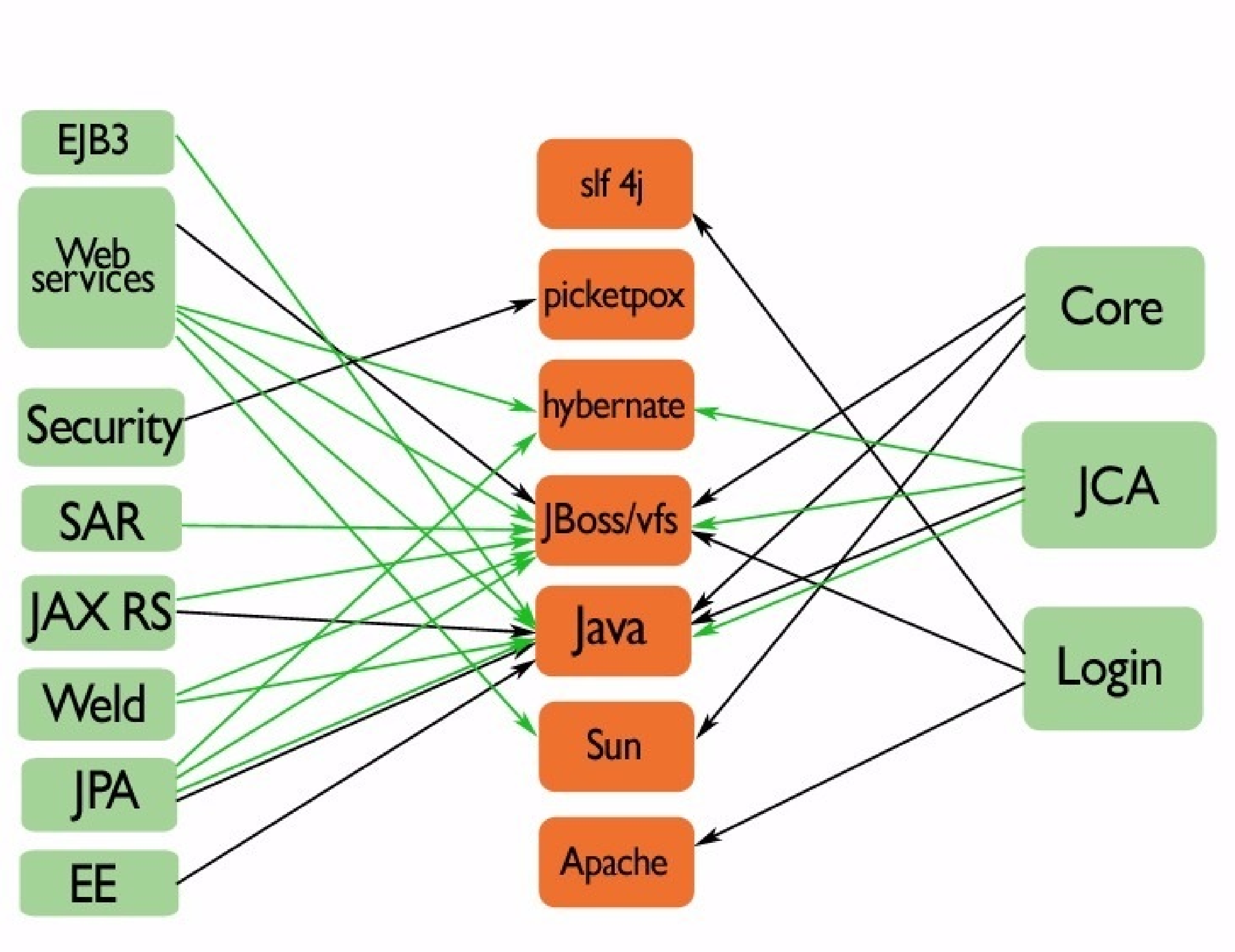


From the analysis we conducted on both pull requests and issues, we found out that the Stakeholders which are involved are the Developers. The Developer class, could be further divided in leading developer(for each component of the system), common developer and Open source community. Each one in this category, has different abilities. The leading developer is responsible to review the code of pull requests and to decide which pull request is going to be merged or not. They make merge decisions based on the quality of code, the style and if it fits the project's roadmap. If a pull request is not merged the leading developers usually send a brief explanation or provide a log file. The common contributors(of the system) are responsible to resolve different issues (e.g bugs, feature upgrades, feature requests) and the open source community developers can assist to resolve open issues by submitting pull requests.

Technical Environment

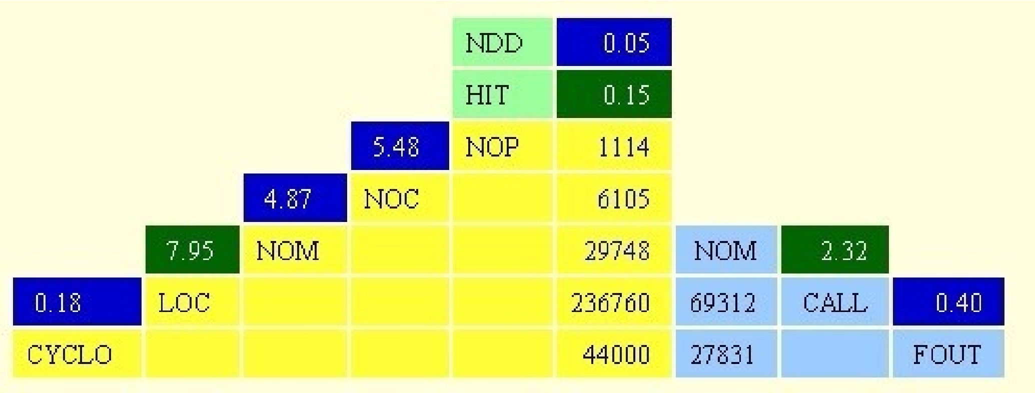
WildFly is a server application that provides easy communication between the user and its server. It is one of the many features of the JBoss Application Enterprise. WildFly provides remote management for one or more servers and runs on a local host server as well. This includes domain mode, where a domain of servers on different hosts can be managed by a central domain controller.

The system is organized in modules. The user can activate only the features that are useful for him. Activating a feature most of the times comes with added dependencies. This status, described in the WildFly official documentation [22], is described in the diagram below. Green arrows represent dependencies that exist only when the feature is activated, where the black ones are constant dependencies. The green boxes are WildFly modules described in the documentation and the orange ones external sources.



System Quality

We looked into the quality of the project using iPlasma [20] and inCode [21]. The project has over 700 000 JavaLOC. Overall the class hierarchies have an average height and the inheritance trees are narrow. The classes are rather small and organized in fine-grained packages. These are all signs of a good system since the JBoss development team wants a modular system. The methods have average length and have simple logic, with few conditional branches. They also call several methods from few other classes. Making this a good environment for new open source developers. The overview pyramid "is an integrated, metrics-based means to both describe and characterize the overall structure of an object-oriented system, by quantifying the aspects of complexity, coupling and usage of inheritance"[28]. The overview pyramid in itself shows that the JBoss developers take care of their technical debt.



				NDD	0.05			
				HIT	0.15			
			5.48	NOP	1114			
		4.87	NOC		6105			
	7.95	NOM			29748	NOM	2.32	
0.18	LOC				236760	69312	CALL	0.40
CYCLO					44000	27831		FOUT

The JBoss developers opted for test driven development [23]. They have a comprehensive test suite. This shows interest in maintaining a qualitative system with the smallest amount of technical debt. To this end we found on the Jira JBoss account they have reported 7,555 issues on duplication, 3,140 issues on refactoring, 220 issues on dead code and 74 issues with misplaced code to name a few [24].

During our analysis we found potential candidates for refactoring. For our first PR [25] we resolved their worst case of duplicated code. A 65 line method in 6 different test cases. We have created this simple issue ourselves [26] for our initial contact. A more detailed description of this process can be found in the contributions file. In the following weeks we plan to tackle some of their God Classes.

System Architecture

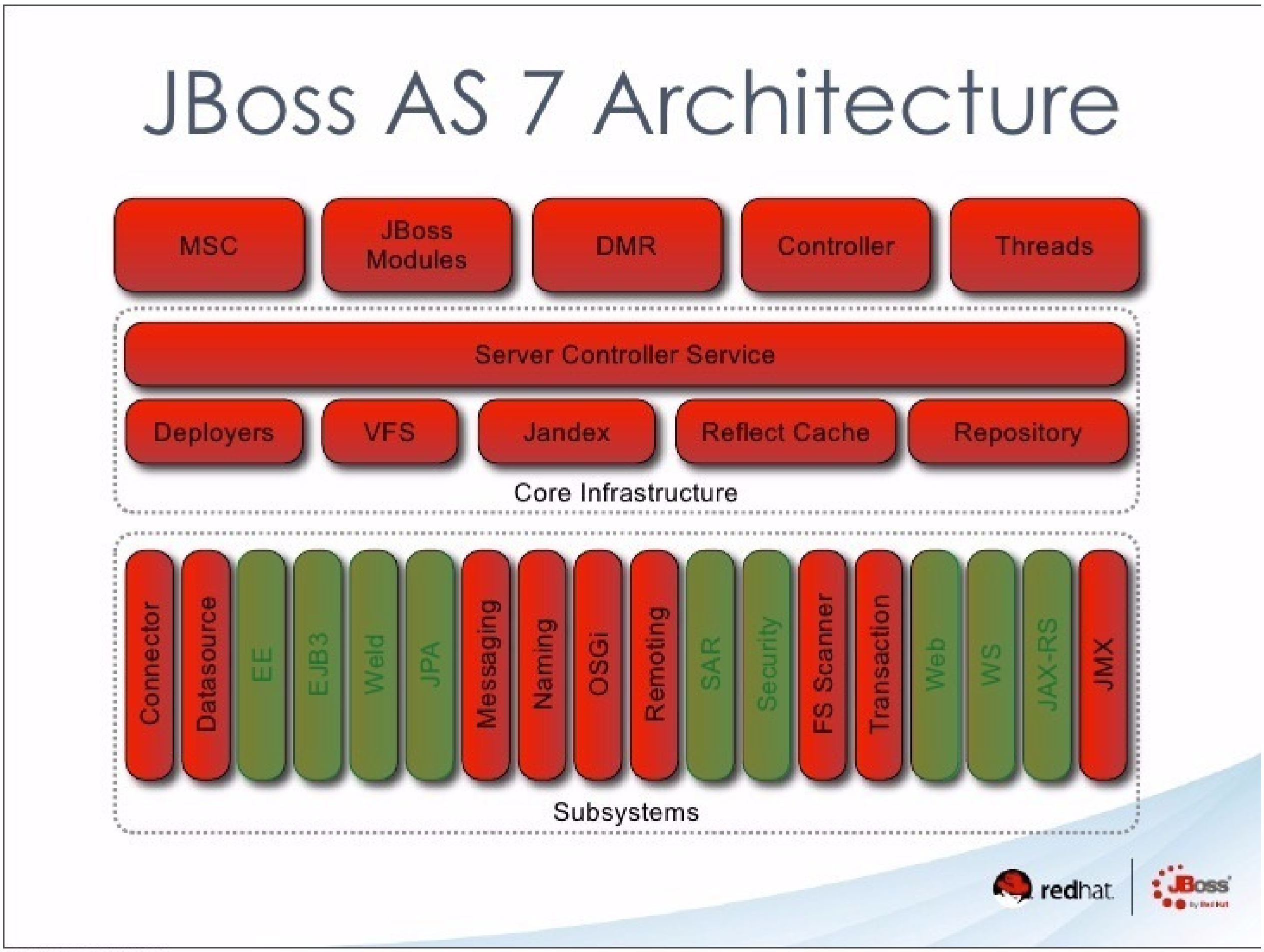
From the beginning (JBoss) had a new approach to the architecture, a "modular class loading and a dependency injection framework allowing the services providing the application server functionality to be installed in parallel", as described by Kabir Khan, five year old software engineer on this project [19]. Throughout the version releases, the modules and their interaction have changed in order to improve the system.

WildFly modules are isolated by default. Some dependencies of modules defined by the application server are set up automatically. These dependencies can be divided in four categories:

- System Dependencies
- User Dependencies
- Local Resources
- Inter Deployment Dependencies

Each one of the categories refers to a specific type of dependencies. The System dependencies refer to dependencies that are added to the module automatically, like Java Api's. The user dependencies are the dependencies which are added through JBoss deployment (not open source). Local Resources are packages inside the deployment like class files from WEB-INF or WEB-INF/lib. Finally, the inter deployment dependencies are the dependencies which affect other deployments in an EAR deployment.

In order to create the development view we researched WildFly official documentation, source code from GitHub and documentation from previous versions. We found the architecture of JBoss AS 7 which is the previous name of WildFly. The JBoss AS 7 was renamed into WildFly 7, the current version is WildFly 10. The green boxes are subsystems that were kept up to version 10.



JBoss AS 7 Architecture

MSC
JBoss Modules
DMR
Controller
Threads

Server Controller Service
Deployers
VFS
Jandex
Reflect Cache
Repository
Core Infrastructure

Connector
Datasource
EE
EJB3
Weld
JPA
Messaging
Naming
OSGi
Remoting
SAR
Security
FS Scanner
Transaction
Web
WS
JAX-RS
JMX
Subsystems

Sunday, June 19, 2011

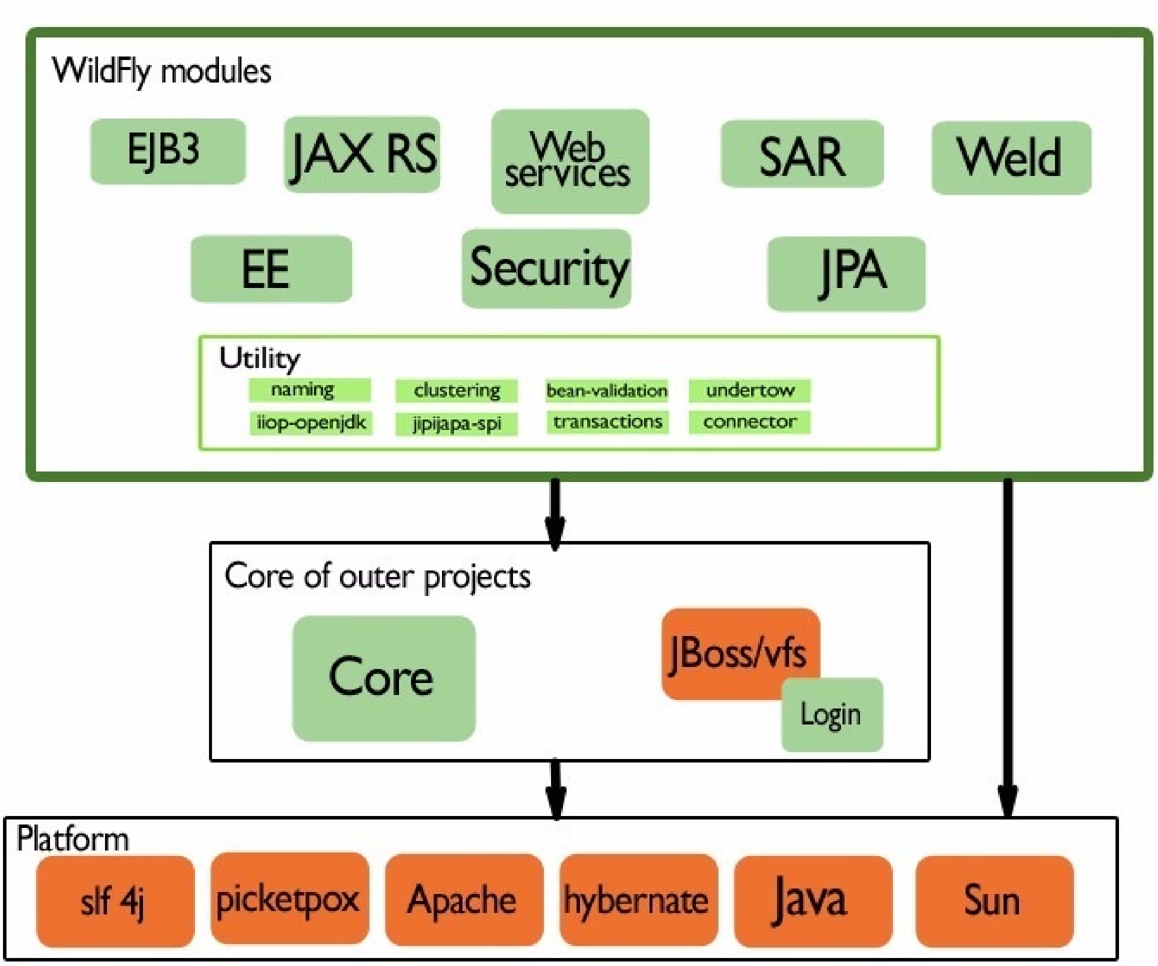
By comparing the architecture of WildFly 7 (JBoss AS 7) [29] to the current code we found some very interesting insights. Alterations have been made through the versions of WildFly like merging of subsystems, adding new functionalities and changing the hierarchical order.

The WildFly project we are looking into contains the modules the user can plugin into the core system. The core is a separate GitHub project [27]. The WildFly project under consideration contains 38 maven projects. Out of these, there are 8 modules the user can plugin, the remaining 30 modules form the WildFly Utility layer, that a user can set up, which provides services for all the main feature modules. Each one of the 8 modules has an explicit dependency on junit for testing purposes. Also, in the WildFly.core project there is a subsystem which contains several tests for the project. These tests refer to several aspects of WildFly like:

- Core
- Api
- Cluster
- Domain
- Integration
- Spec

This six sets of tests aim to assess any change on the initial code, in order to avoid any unwanted situation, like internal errors, bad requests, systems' malfunctions, etc.

The main user enabled functionalities (green large WildFly modules Figure 9) are dependent on the core and JBoss libraries (i.e. Login). They are also indirectly or directly dependent upon the external sources that form the platform layer of our development view. The rest of the project forms the utility layer that contains the features the modules need. Each module has its own custom mix of dependencies that we traced from the pom files in the project. This overview can be seen in the following diagram:

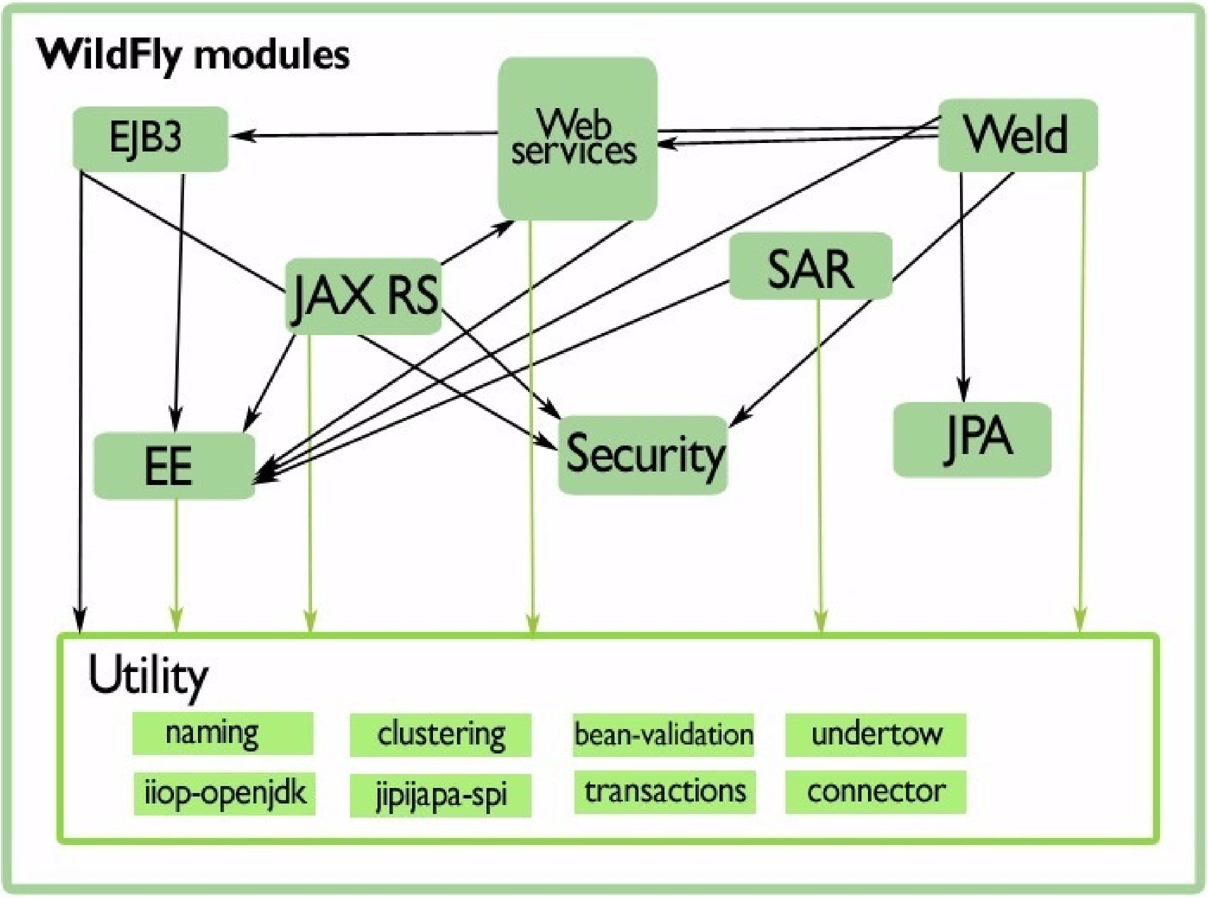


The main user enable functionalities are:

- **EJB3**: The Enterprise JavaBeans architecture or EJB for short is an architecture for the development and deployment of component-based robust, highly scalable business applications. These Applications are scalable, transactional, and multi-user secure.
- **Web Service**: JBossWS is a web service framework developed as part of the JBoss Application Server / WildFly. JBossWS integration provides the application server with any WS related technologies it needs for achieving compliance with the Java Platform.
- **Security**: The security subsystem is the subsystem that brings the security services provided by PicketBox to the JBoss Application Server 7 server instances.
- **SAR**: these deployment packages are just JAR files with special XML deployment descriptors in directories like META-INF and WEB-INF, which allows you to make changes to web pages etc on the fly without re-deploying the entire application.
- **JAX RS**: RESTEasy is a JBoss project that provides various frameworks to help build RESTful Web Services and RESTful Java applications. It is a fully certified and portable implementation of the JAX-RS specification. JAX-RS is a new JCP specification that provides a Java API for RESTful Web Services over the HTTP protocol.

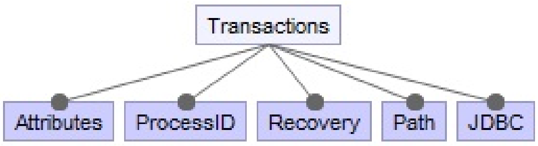
- **Weld**: Weld is the reference implementation of CDI: Contexts and Dependency Injection for the Java EE Platform which is a JCP standard for dependency injection and contextual life-cycle management and one of the most important and popular parts of the Java EE platform.
- **JPA**: The Java Persistence API (JPA) is a Java specification for accessing, persisting, and managing data between Java objects / classes and a relational database. JPA was defined as part of the EJB 3.0 specification as a replacement for the EJB 2 CMP Entity Beans specification.
- **EE**: The EE subsystem provides common functionality in the Java EE platform, such as the EE Concurrency Utilities (JSR 236) and injection. The subsystem is also responsible for managing the life-cycle of Java EE application's deployments, that is, .ear files.

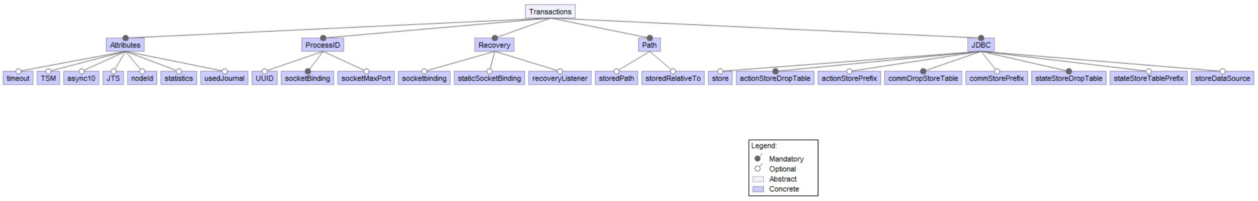
Even though the modules should be independent of each other, we highlight their inter dependencies in the following zoomed in diagram:



Transaction Feature System Analysis

We focus on the Transaction user customization, specific to the WildFly project in discussion. This is a maven project in itself, part of the utility layer presented in the architecture overview. The transaction has five sections with a total of 23 flexibility points described as follows.





1. Attributes
    - Timeout: A typical transaction might be started by a Session. If the duration of these transactions exceeds the specified timeout setting, the transaction service will roll-back the transactions automatically.
        - Enable TSM Status: Whether the transaction status manager (TSM) service, needed for out of process recovery, should be provided or not.
        - Journal Store Enable Async: Whether AsyncIO should be enabled for the journal store. Default is false. The server should be restarted for this setting to take effect.
    - JTS: If true this enables the Java Transaction Service.
        - Node Identifier:
          Used to set the node identifier on the core environment.
        - Statistics Enabled: Whether statistics should be enabled.
    - Used Journal Store: Used for writing transaction logs. Set to true to enable and to false in order to use the default log store type. The default log store is normally one file system file per transaction log. The server should be restarted for this setting to take effect. It's alternative to jdbc based store.
2. Process ID
    - UU ID: Indicates whether the transaction manager should use a UUID based process id.
        - Socket Binding: The name of the socket binding configuration to use if the

transaction manager should use a socket-based process id. Will be 'undefined' if 'process-id-uuid' is 'true'; otherwise must be set.

- Socket Max Port
The maximum number of ports to search for an open port if the transaction manager should use a socket-based process id. If the port specified by the socket binding referenced in 'process-id-socket-binding' is occupied, the next higher port will be tried until an open port is found or the number of ports specified by this attribute have been tried. Will be 'undefined' if 'process-id-uuid' is 'true'.

3. Recovery
    - Socket Binding

      Used to reference the correct socket binding to use for the recovery environment.
    - Statics Socket Binding

      Used to reference the correct socket binding to use for the transaction status manager.
    - Recovery Listener

      Used to specify if the recovery system should listen on a network socket or not.
4. Path
    - Object Stored Path Denotes a relative or absolute file-system path denoting where the transaction manager object should store data. By default the value is treated as relative to the path denoted by the "relative-to" attribute.
    - Object Stored Relative to References a global path configuration in the domain model, defaulting to the JBoss Application Server data directory (jboss.server.data.dir). The value of the "path" attribute will treated as relative to this path.
5. JDBC
    - Store Use the jdbc store for writing transaction logs. Set to true to enable and to false to use the default log store type. The default log store is normally one file system file per transaction log. The server should be restarted for this setting to take effect. It's alternative to Horneq based store
    - Action Store Drop Table Configure if jdbc action store should drop tables. Default is false. The server should be restarted for this setting to take effect.
    - Action Store Prefix Optional prefix for table used to write transaction logs in

configured jdbc action store. The server should be restarted for this setting to take effect.

- Communication Drop Store Table Configure if jdbc communication store should drop tables. Default is false. The server should be restarted for this setting to take effect.
- Communication Store Prefix Optional prefix for table used to write transaction logs in configured jdbc communication store. The server should be restarted for this setting to take effect.
- State Store Table Prefix Configure if jdbc state store should drop tables. Default is false. The server should be restarted for this setting to take effect.
- Jdbc state store table prefix Optional prefix for table used to write transaction logs in configured jdbc state store. The server should be restarted for this setting to take effect.
- Store Data Source Jndi name of non-XA data source used. Data source should be define in data sources subsystem. The server should be restarted for this setting to take effect.

Transaction Features Management

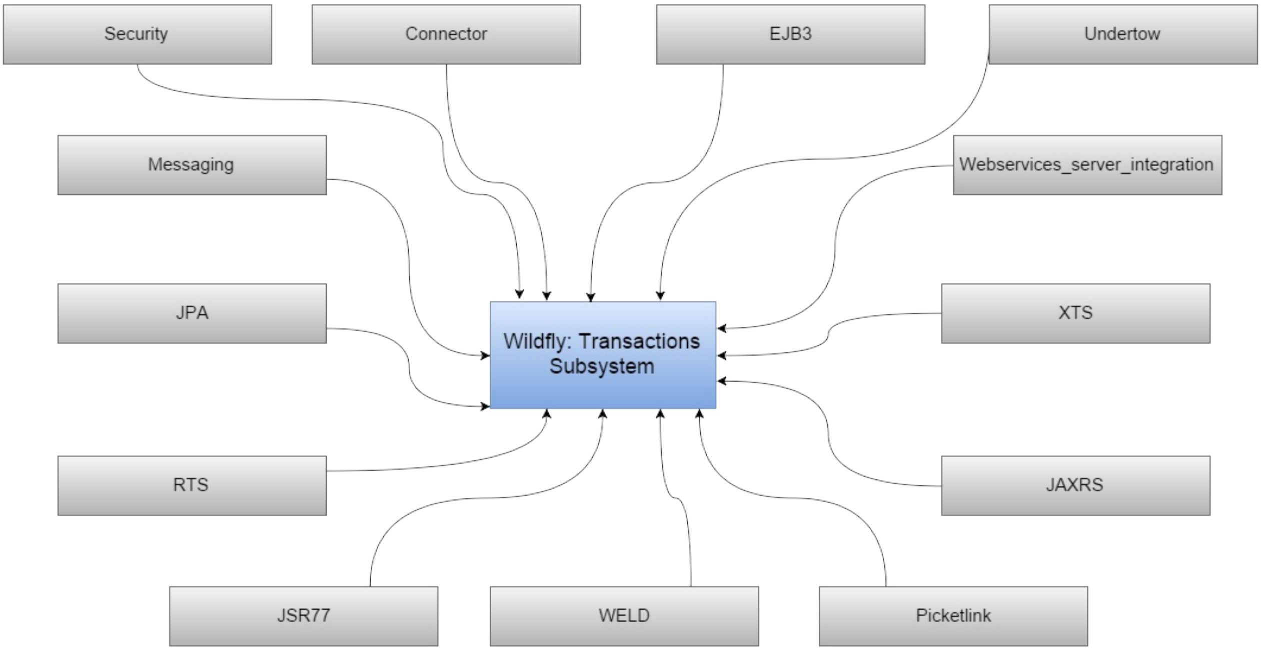
The transaction features are vital to the system. That is why a valid mix of mandatory transaction features are stored for default settings in a configuration xml file. The whole feature processing algorithm starts by calling the start method from CoreEnvironmentService class by the Runner. Their approach makes use of a JBoss closed implementation of a CoreEnvironmentBean and a SocketBinding. The entire process has a roll back option that can give updates on the current status.

The transaction process is dependent upon the JMX and EE (i.e. Concurrent's TransactionSetupProviderService) subsystems.

In regards to configuring these feature we have found 7,832 Jira issues on transaction, 201 on journal store, 87 on Object Stored Path and 39 on Statics Socket Binding to name a few.

Transaction Features Dependencies

The outcome of transaction feature selection has a direct impact in the variability of the custom user instances of the WildFly system. At the same time the transaction module is used by other modules as shown in the diagram.



This means that all the developers of the modules dependent upon transactions are affected by these features. The transaction module in WildFly monitors all the incoming and out-coming communications of the server. We can observe that the transaction subsystem is vital for the proper flow of almost all the main components of WildFly.

Based on the source code and our experience as a mock user we identified the following dependencies between the transaction features.

1. `Journal store enable async io` **with** `Use journal store`

   > In order to set the `Journal store enable async io` to TRUE first you have to enable the `Use journal store`

2. `ID UUID` **with** `ID socket binding`

   > If the `ID UUID` is disable then you **MUST** assign as an id in `Id socket binding`

3. `Id socket binding` **with** `Process id socket MAX port`

   > The `Process id socket MAX port` can **NOT** be assigned if the `id socket binding` is not defined

4. `Use JDBC store` **with** `JDBC store datasource`

   > If you enable `JDBC store` you have to provide a `JDBC store datasource`

5. `Process id socket MAX port` **with** `ID UUID`

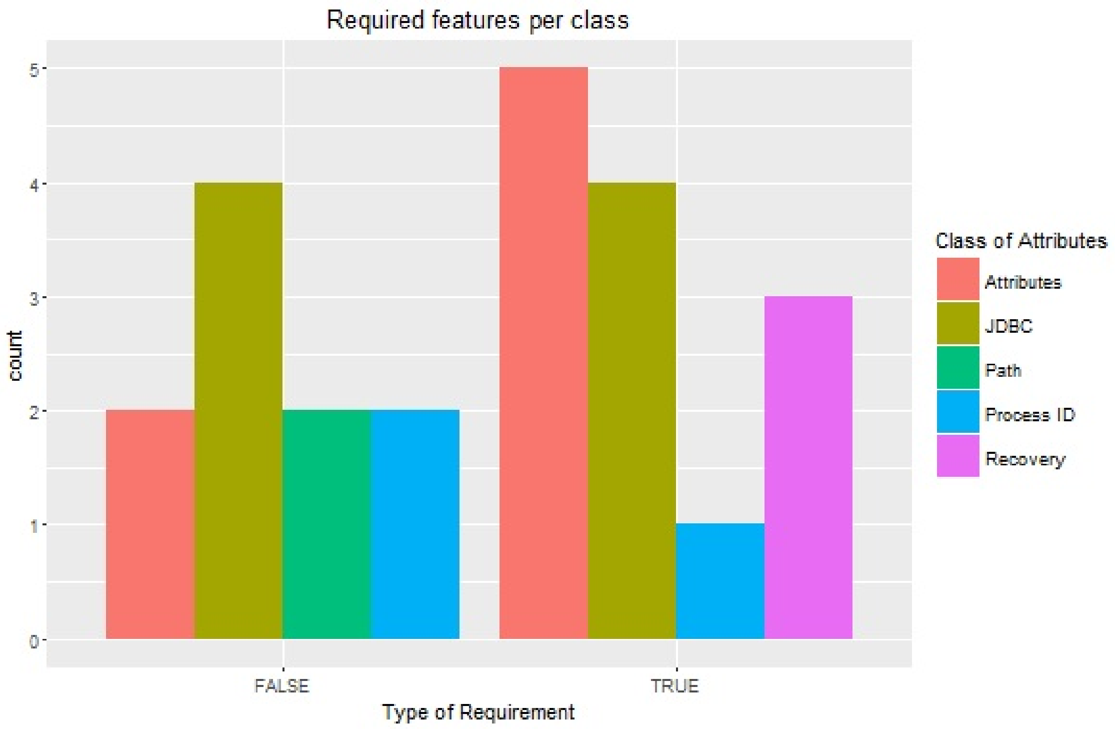
   > If the `ID UUID` is true the `Process id socket MAX port` must be undefined

Binding Time

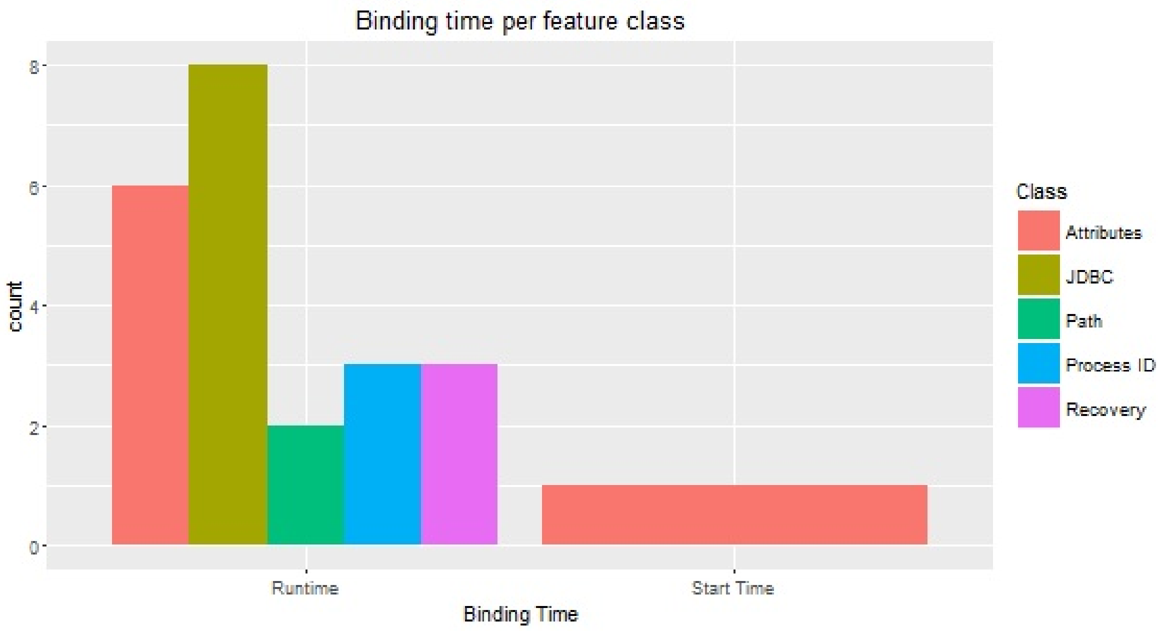
We also categorized each feature to a certain class and binding time.

Features	Class	Required	Binding
Timeout	Attributes	FALSE	Start time
Enable TSM Status	Attributes	TRUE	Runtime
Journal Store Enable Async IO	Attributes	TRUE	Runtime
JTS	Attributes	TRUE	Runtime
Node Identifier	Attributes	FALSE	Runtime
Statistics Enabled	Attributes	TRUE	Runtime
Used Journal Store	Attributes	TRUE	Runtime
UU ID	Process ID	TRUE	Runtime
Socket Binding	Process ID	FALSE	Runtime
Socket Max Port	Process ID	FALSE	Runtime
Socket Binding	Recovery	TRUE	Runtime
Statics Socket Binding	Recovery	TRUE	Runtime
Recovery Listener	Recovery	TRUE	Runtime
Object Stored Path	Path	FALSE	Runtime
Object Stored Relative to	Path	FALSE	Runtime
Store	JDBC	TRUE	Runtime
Action Store Drop Table	JDBC	TRUE	Runtime
Action Store Prefix	JDBC	FALSE	Runtime
Communication Drop Store Table	JDBC	TRUE	Runtime
Communication Store Prefix	JDBC	FALSE	Runtime
State Store Table Prefix	JDBC	TRUE	Runtime
Jdbc state store table prefix	JDBC	FALSE	Runtime
Store Data Source	JDBC	FALSE	Runtime

The following figures illustrate a few examples of the relationship between the class, binding time and the required features. The Attributes and JDBC class have the most required attributes. Also, JDBC has an equal number of required and optional features.



An overview of the feature binding time and class shows that only the *Default timeout* feature, which defines the timeout of the server requires reloading the server (i.e. restart). The rest of the features can be modified at Runtime.



Further Reading

1. Install/Upgrade/Debugging Steps
2. Demo: Mail configuration
3. Perspectives

- Evolution
- Development

4. Viewpoint
    - Operational
    - Deployment

Conclusion

WildFly is an open source server management application. We analyzed the architecture, stakeholders, development context and feature implementation, all detailed in this chapter. We also provide a further reading section that links to more resources developed by our team. The WildFly system is split into the core that contains the basic functionalities and the WildFly project that contains all the user enabled modules.

The quality of the code is good but still contains dependencies between modules that should be independent. We tried to contribute to a more modular architecture by our PR`s, by clearing duplicated code and a feature envy bad smell. The contribution procedure is very complex making it hard to get merged. We did however officially published a tutorial for installing, debugging, updating WildFly and a demo. The tutorial was our natural reaction to the available online information on WildFly. Based on our study we consider this system to be very well maintained, developed and tested.

References

1 http://www.redhat.com/en/about "Red Hat"

2 https://en.wikipedia.org/wiki/WildFly "Wildfly_wiki"

3 http://wildfly.org/about/ "Wildfly"

4 https://www.redhat.com/en/technologies/jboss-middleware/application-platform "Jboss"

5 https://en.wikipedia.org/wiki/JBoss_(company) "jboos_wiki"

6 "Rozaski and woods.software architecture"

7 http://hibernate.org/ "hibernate"

8 http://narayana.io/ "narayana"

9 http://infinispan.org/ "infispan"

10 https://en.wikipedia.org/wiki/Red_Hat "Redhat_wiki"

11 https://www.linkedin.com/in/arunpgupta "arunpgupta"

12 http://blog.arungupta.me/wildfly-cluster-raspberrypi-techtip28/ "cluster_rasp"

13 https://www.youtube.com/watch?v=5MhqcVFVJ2s "youtube"

14 http://www.cafepress.com/jbossorg/6726696 "cafepress"

15 https://www.linkedin.com/in/ankur25 "ankur25"

16 https://docs.jboss.org/author/pages/viewpreviousversions.action?pageId=80873103 "jboss_previous_version"

17 http://wildfly.org/downloads/ "wildfly_downloads"

18 http://awestruct.org/ "awestruct"

19 https://www.linkedin.com/in/kabir-khan-99a333 "Kabir Khan Profile"

20 http://loose.upt.ro/reengineering/research/iplasma "iPlasma"

21 https://www.intooitus.com/products/incode "inCode"

22 https://docs.jboss.org/author/display/WFLY8/Documentation "WildFly Documentation"

23 https://developer.jboss.org/wiki/TestDrivenDevelopmentTDDAndMovingForwardWithLessTechnicalDebt "jboos TDDi"

24 https://issues.jboss.org/secure/Dashboard.jspa "Jira JBoss"

25 https://github.com/wildfly/wildfly/pull/8740 "PR1-github"

26 https://issues.jboss.org/browse/WFLY-6296?jql=project%20%3D%20WFLY%20AND%20resolution%20%3D%20Unresolved%20ORDER%20BY%20priority%20ASC%2C%20updated%20DESC "PR1-JIRA"

27 https://github.com/wildfly/wildfly-core "core project"

28 Lanza, Michele, and Radu Marinescu. (2007) *Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems.* Springer Science & Business Media.

29 http://www.slideshare.net/rayploski/jboss-application-server-7

Youtube-dl

ROB VAN BEKKUM
RVANBEKKUM

JORDEN VAN BREEMEN
JORDENVBREEMEN

STEVEN GOSSELING
STEVENGOS

SANDER VAN DEN OEVER
SANDERVDO

Abstract

Youtube-dl is a simple command-line application that enables users to download videos from one of the more than seven hundred supported websites. The application is actively maintained and developed by a relatively small group of developers. On a regular basis the project receives external contributions and requests (e.g. for supporting a new site) from developers and users. The simplicity of the architecture, consisting of four major components, allows for easy, continuous integration of new functionality. The application offers various features in the form of options that allow for retrieving specific data, extensive (downloading) configurations and post-processing tasks. Finally there are some regulations (among others regarding *copyright infringement* and the *Terms of Service* of websites) that the application has to deal with and take into account.

Table of Contents

Introduction

Development of youtube-dl has been started over three years ago by Ricardo García, the original owner of the repository. He stepped down a while ago as the maintainer of the repository and it is now maintained by a community of active developers. In an earlier stage of development the architecture made it hard to extend the functionality of the application. At that point the architecture was completely redesigned to allow for easy integration of new site support.

Youtube-dl is a command-line application. However, graphical user interfaces are available as extensions to the application. Users can provide the URL of a website where they have observed the video (or other supported content) and youtube-dl will attempt to download the media located at that URL. The user can specify additional options to customize the behavior of youtube-dl. The simplistic foundation and the Python programming language enable the application to run on all most common operating systems.

Currently the application is being contributed to nearly every day. The active community and the extensible architecture allow for easy integration of new functionalities. As a result current users are able to download media content from more than seven hundred different sources. The application also supports downloading entire playlists, embedding subtitles, extracting audio and much more (see also the section on Features).

This chapter aims to give users insight in how this application is developed, maintained and used. This way one will have a quick understanding of youtube-dl. First the stakeholders and the context view are used to provide insight in the organization of youtube-dl. This is followed by an in-depth analysis of the architecture of the application in order to understand its structure. Afterwards, the wide variety of features available in the application are discussed followed by a short discussion about the regulations the application and the users might face. Finally, there is a short conclusion regarding our findings about the project.

Organization

This section will discuss the involved parties during the development, maintenance and execution of the application. These parties are then visualized using a context-view diagram.

Stakeholders

Youtube-dl is maintained under the public-domain license. It does not have a foundation nor company attached to it that is involved in backing or supporting the maintenance or development of the application. Besides, no direct commercial opportunity is available for youtube-dl in the current situation. The most important stakeholders that are identified based on the categorization of Rozanski and Woods [1] are discussed below.

The **users** of youtube-dl consist of all users that use the application to download content from one of the supported websites. The application is most commonly used by non-commercial users that wish to obtain videos for offline usage. Commercial users of the application consist of websites that re-host videos published on one of the supported websites. There are, however, no commercial parties that acknowledge that they specifically use youtube-dl for such purposes.

The current most active **maintainers** of the project consist of the following members of the community: Philipp Hagemeister, Sergey M., Jaime Marquínez Ferrándiz and Yen Chi Hsuan. These members are all actively involved in the development and have collaborator access to the project on GitHub. Therefore, they oversee and review all new contributions to assure the quality of the project, and hence can be considered as both **assessors** and **integrators**.

A complete list of people who are credited for their contributions to the project is available in the project repository. Every time a reasonably large contribution is made to the project one is asked if he/she wishes to be added to this list. Before a merge can take place developers are required to **test** the new functionality themselves and the code is inspected by one of the maintainers. The core development is mostly done by the four maintainers mentioned earlier, who also have the most influence on which pull requests get merged. However, another community member that holds the nickname remitamine also contributed a lot of development work in recent releases. He has also been granted a collaborator status. What is peculiar here is that Ricardo García is the founder and owner of the project but announced in his blog and on Hacker News that in 2011 he has stepped down as maintainer of youtube-dl, though still owns the repository and the website.

The community offers **support** using the GitHub issue system. This support is provided by the community of youtube-dl, here different kind of stakeholders ask for support, ask questions and they submit bug reports and feature requests. Support at issues is often provided by one of the maintainers, though sometimes other community-members get involved.

In the recent support issues the most involved member is Sergey M.. He is often involved in answering questions from the community or taking care of bug reports and missing functionality. He could therefore, apart from **developer**, be considered as a main **support**

staff member as well. The application is distributed using both GitHub and the website from Ricardo García, these could be considered the **suppliers** of the application.

Other involved parties are the websites that are supported by youtube-dl. Some of these websites will have commercial applications for the hosted videos and thus have rules that specify that downloading the videos from their website is not allowed. The youtube-dl application could effect their services and break the policy for these websites. This could have a negative effect on both the supported websites and the legal position of youtube-dl. We will discuss this in more detail within the Regulations section.

Next to these parties there are applications available that use youtube-dl as a foundation/extension or dependency for their own application. Some examples are youtube-dl-gui and Instant Music Downloader, which rely on the functionality of youtube-dl for their application to function properly.

Finally, youtube-dl has to deal with some **competitors** like DVDVideoSoft that deliver (partly) the same functionalities that youtube-dl delivers. These stakeholders are involved as they could take away some clients from the application.

Context View

To give a view of the context in which youtube-dl resides, we identify the scope of the application and describe the interactions and dependencies between the application and its environment. Therefore we consider the external services and data that youtube-dl depends on to enable it to do what it is build for.

Functional Capabilities

The most important functionality that youtube-dl offers is downloading content from any of the supported platforms. Youtube-dl supports these platforms by having a separate Information Extractor (IE) implemented for each platform. The software architecture is designed in such way that it focuses on the process of adding support for new platforms by adding a new IE being a relatively simple tasks for developers.

The core of youtube-dl implements the other functionalities, which are:

- **Parsing the input and options**: Enabling users to download content from all supported platforms in a uniform way.
- **Downloading files from the media platforms**: Actually downloading the content is handled by youtube-dl and is independent of the media-hosting platform.
- **Post-processing the downloaded content**: Processing files in a certain way defined by options set by the user.

Youtube-dl also accounts for other features such as downloading content that requires authentication, retrieving subtitles and setting other more advanced options (e.g., network configurations). The application can be used in a uniform way for downloading from various platforms where it uses IEs to decide how to extract the required data. It therefore is not concerned with automatically identifying the platform and deciding how to retrieve the content from it.

External Entities and Interfaces

There are multiple external entities that youtube-dl interacts with, the first we identify being the platforms such as; YouTube, Vimeo or SoundCloud, from which the application extracts data and retrieves content (e.g., video and audio files) that are requested by its users. The data the application depends on for downloading this content is extracted using an IE for the specific platform. Based on a URL of the content provided by the user, this IE fetches a dictionary of information about the content that is to be retrieved that is subsequently used by youtube-dl to download the content from the platform.

The community of youtube-dl is primarily active on GitHub. As mentioned earlier it is used for issue tracking. Other than that it is used for (external) contributions of developers and communicating the contribution guidelines and information about the usage of the application. Contacting the core developers of the system is also possible through a dedicated IRC channel on Freenode. The application uses the Travis CI system for continuous integration, but seems to have abandoned the active use of the system at the time of writing for over two years.

Furthermore, youtube-dl has a built-in testing system which makes use of the Tox testing tool for Python for testing each of the IEs of the application. In order to contribute by adding support for a new IE, developers are required to write at least a single test for the IE, which typically comprises of downloading the content for a sample URL. Next to this, developers are required to use the Flake8 tool to check for adherence to the PEP8 style conventions, logical errors and code complexity. Finally, developers can contribute to youtube-dl by making a pull request which then will be reviewed by the maintainers.

In Figure 1 a visual overview is provided of the interactions of youtube-dl with external entities and capabilities and characteristics of the application. This diagram serves as a high-level view of the system not considering the detailed structure or implementation of the system.

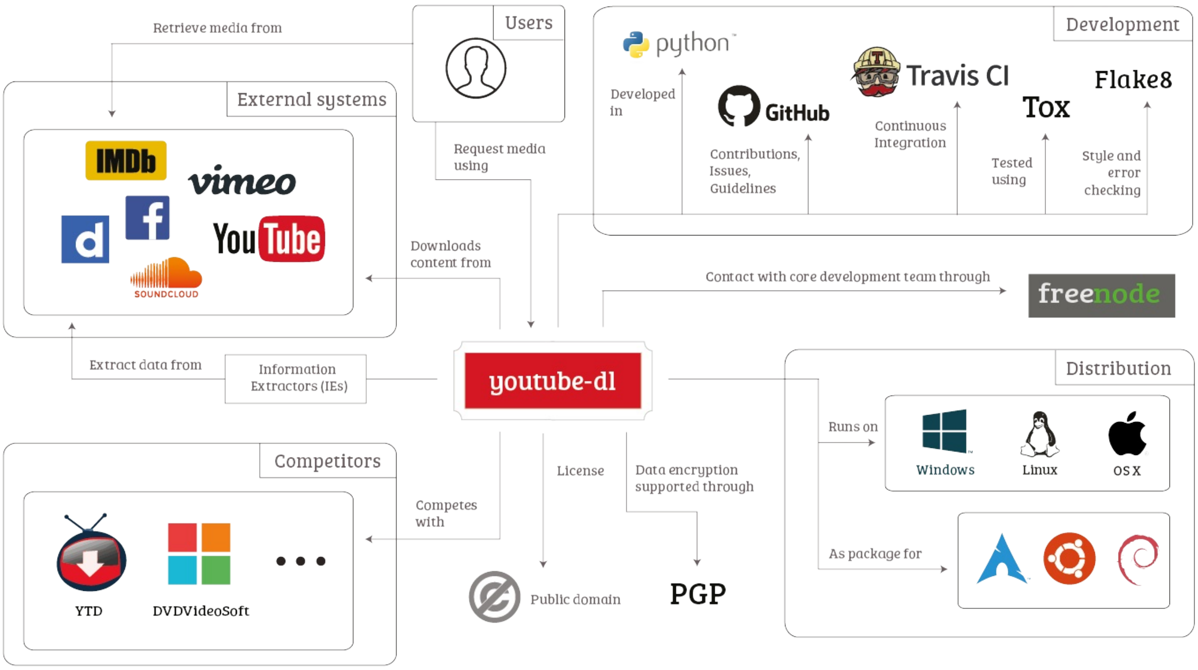


Figure 1 - *A visual representation of the context view of youtube-dl.*

Impact

Since everyone is free to use, alter and distribute the code of youtube-dl a possibility is to integrate youtube-dl in another application (like a GUI wrapper). These tools depend on the data format that youtube-dl requires. As soon as this format will change, the wrappers will most likely require some changes too. Given the fact that the command structure has not changed for over a year based on the blame log it is not likely to expect changes there anytime soon. Only some parameters changed slightly and in most cases there was only new instead of changed functionality. This provided some certainty for applications that integrate youtube-dl as it will not likely break these applications when youtube-dl is being updated.

Architecture

In this section we aim to give the reader more insight into the architecture of the application. The component structure of the application is explained and visualized using a diagram. Next the information flow between the different components is discussed, describing the conditions under which the different types of information are exchanged. Finally we take a look at how the development of the application takes place.

Component Structure

Youtube-dl consists of a couple of components. These components and their relationships are visualized in Figure 2. The main components are:

- **YoutubeDL**: the core of the application. This component is responsible for the overall process. It processes the input, parsed arguments from the command-line and information from assumptions made by youtube-dl itself.
- **Extractors**: responsible for gathering the information about the video. The extractors are able to extract video-urls from webpages through analysis (regular expressions for instance). The extractors provide the information in a pre-defined format so that it can be processed further by the *YoutubeDL* component.
- **Downloaders**: the downloaders are able to transfer a remote video to the local filesystem. Depending on the media-format this component will determine which downloader to use.
- **Post Processors**: responsible for any post-download operations that should be applied on the video. Think of embedding subtitles, extraction the audio, etc.

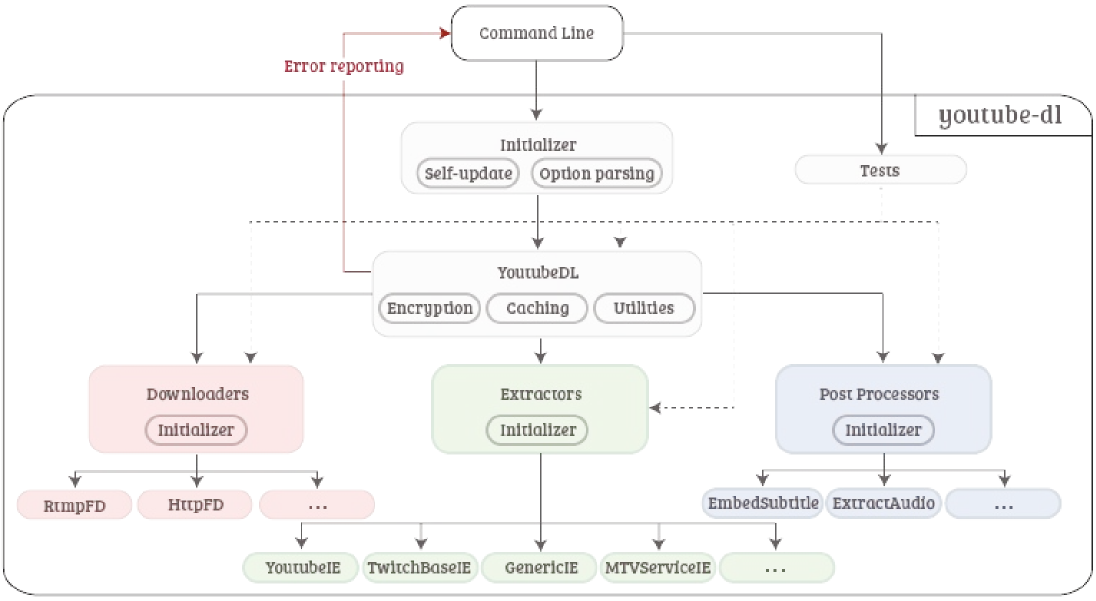


Figure 2 - *Component structure of youtube-dl.*

Default Program Flow

As can be seen in Figure 2, we have the core component (*YoutubeDL*) that controls the process. When the application is started the initializer (depicted at the top) will perform a self-update (if requested by the user) and it will parse the arguments from the command-line. Then *YoutubeDL* requests an extractor by using the initializer of the *Extractors* component which helps determining which extractor to use. As soon as *YoutubeDL* receives back the required information it can pass this information to the downloaders. The initializer of the

Downloaders component then determines which downloader to use (depending on the file type/properties) and tries to download the corresponding file. Finally, *YoutubeDL* applies post-processors according to the user's wishes. The initializer of the *Post Processors* component determines which post-processors should be ran and returns the output of the selected processor(s). If the user requested to download multiple files, *YoutubeDL* will proceed and repeat the process all over for the next file.

Test Program Flow

Tests are available for each of the components listed earlier. Tests can be invoked from the command-line. The tests will be retrieved from all the available downloaders, extractors and processors (indicated by the dashed lines in Figure 2). For example, every extractor contains a number of tests. These tests try to extract the information for a given URL. This result is then compared to the pre-defined expected result. It is possible to specify which tests to run. The test suite is growing and therefore the testing process takes longer as youtube-dl evolves. Being able to run only specific tests allows you to quickly test a specific part.

Error Reporting Flow

Youtube-dl aims at centralizing error-reporting. From all the implementations of downloaders, extractors and post-processors the errors are directed to the core, *YoutubeDL*, where the errors are handled. This ensures a consistent way of reporting errors, which makes it a lot easier to maintain the project. *YoutubeDL* in turn prints the errors (in a consistent format) to the command-line (as indicated with the red line in Figure 2).

Extensibility

The structure is designed to be extended easily. Previously the developers encountered problems when they had to 'hack in' additional features. This lead to the decision to implement the application from scratch. In the current implementation there are interfaces for the extractors, downloaders and post-processors. By implementing these interfaces one can easily extend the functionalities of youtube-dl.

Folder Structure

To keep the structure simple, youtube-dl has separate folders for the different kind of implementations. Within the repository there is a folder called `youtube_dl` which contains all of the source-code. Within that folder there are subfolders called `downloader` , `extractor` and `postprocessor` . The related initializers can be found in `__init__.py` for the given subfolder and next to that every subfolder contains a `common.py` which holds the interface that is implemented by all implementations. For instance, each extractor implements the functions declared in the base class in `extractor/common.py`. The filenames of the different implementations are clear, for example the extractor for YouTube videos is located in a source file called `extractor/youtube.py`. A graphical overview can be found in Figure 3.

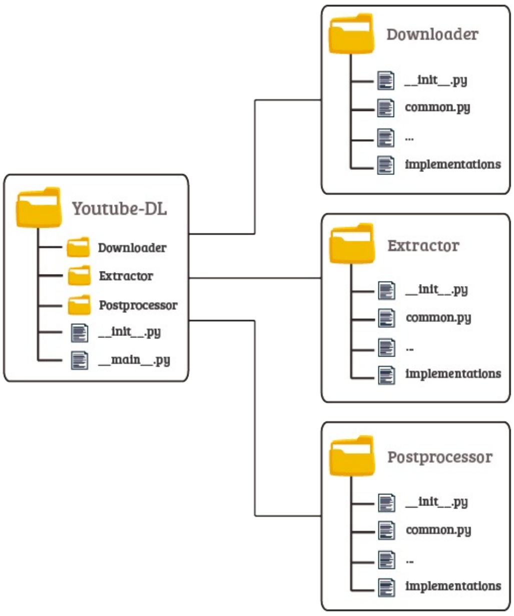


Figure 3 - *Folder structure of youtube-dl. Each of the subfolders houses an initializer, a common interface and several implementations of this interface.*

Information Viewpoint

The flow of information within youtube-dl is not too complex. Visualizing the flow of information exchanged between components at runtime will provide more insight in how the application functions. The flow is best visualized using an information flow diagram as shown in Figure 4.

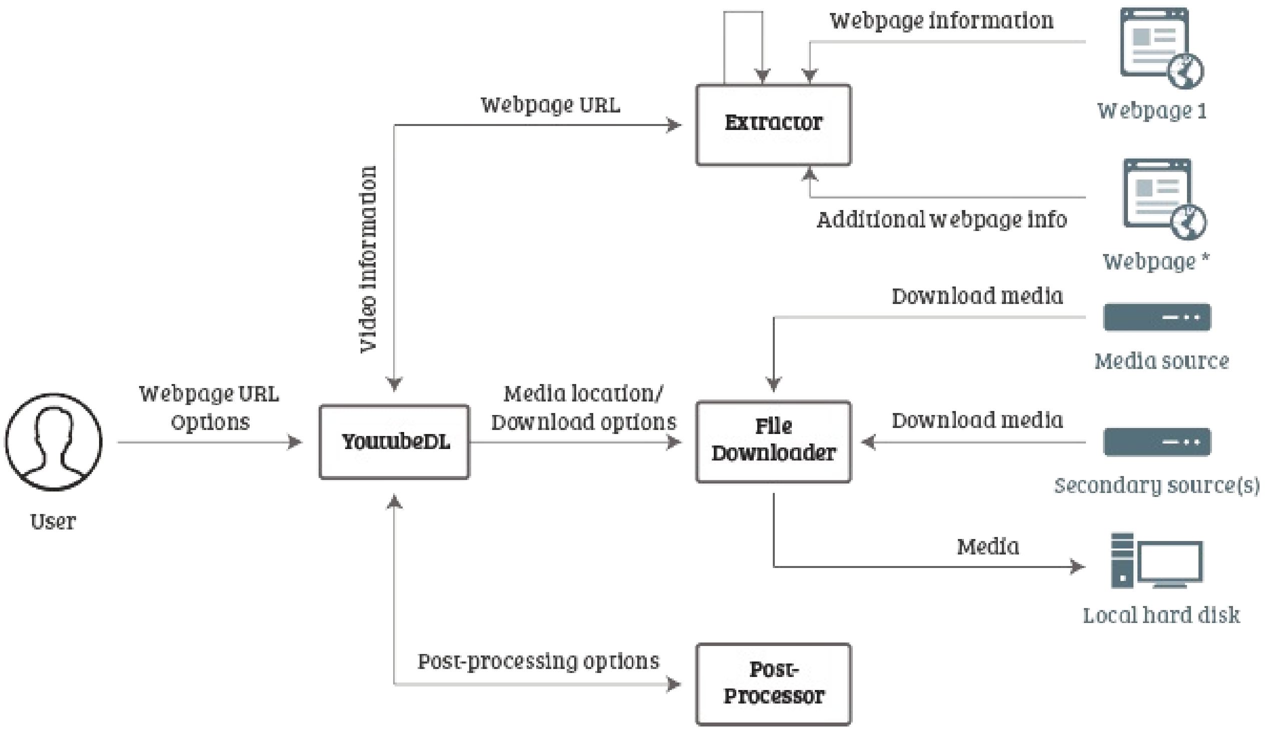


Figure 4 - *Information flow diagram.*

We will discuss the kind of information exchanged between the different components of youtube-dl and under what circumstances this information is exchanged. Initially a user will provide youtube-dl with a URL and some optional options. These options are analyzed and stored in the *YoutubeDL* component. Here options such as the setup of a proxy, age validation or downloading thumbnails are processed.

After initialization the *YoutubeDL* component will create an *Extractor*. The extractor will receive the URL of the webpage and it will return all information that is needed for further processing. The extractor retrieves this information from the webpage and, in some scenarios, from additional webpages that will be downloaded to retrieve (more) information. Whenever a website hosts content for multiple other websites, one can create really small extractors for these websites that all point to the extractor of the hosting website. The extractor for the hosting website will receive the information from the smaller extractors and perform its extraction process. The user will see information about the extraction process on the command-line interface. Which information is shown depends on the extractor and the information that it retrieved from the website.

With the extractor's information the *File Downloader* is capable of downloading data from a specific location. This downloader receives additional information, e.g. information to see whether it has to download a playlist or whether there are bandwidth limitations. This information can originate from the extractor or from the user (through options on the command-line interface). In the scenario of a single file it will fetch the file from its physical location, perform small modifications such as generating filenames, process any options that might influence the file and finally saving the resulting file to the desired location. When the *File Downloader* is called with a playlist option it will retrieve media from multiple media sources that are part of the playlist.

Users can specify options that will (optionally) trigger *Post-Processors*. *Post-Processors* are created in the *YoutubeDL* component and a list of them is stored there. They may invoke a downloader or extractor but this is not executed within the component itself but within *YoutubeDL*. Therefore there is no direct flow of information between the *Post-Processor* and the two other components.

Future executions of youtube-dl are almost completely independent of prior executions. Most information is extracted from the Internet. There is only very few static information stored within youtube-dl. Nearly everything is executed at runtime and stored in memory. The output media is, obviously, a static result and will have no effect on future execution.

Development Process

The development of youtube-dl is open for external contributions on the repository on GitHub. The project relies on the GitHub issue tracker where all bug reports and requests are filed by both users and developers. The guidelines for creating a new issue are documented on the repository and comprises of various questions that should be evaluated before placing an issue. It is possible to obtain logging information at different levels and it is recommended (sometimes even required) by the authors to include the full verbose output in issues to get an overview what happens step-by-step. For several issues the maintainers of the repository attach labels for issues falling within a particular category (e.g., `site-support-request` issues are typically solved by the implementation of a new extractor). Most commonly, external contributors or maintainers create pull requests from the open issues.

Before developers submit their pull requests they should attempt to follow the instructions and adhere to the guidelines and standards as described in the External Entities and Interfaces section. When a pull request is submitted it is reviewed by one of the maintainers who will provide feedback whenever the implementation can be improved or when it can be made more generic (or specific). For example, when adding a new extractor, the maintainers might suggest to add additional tests to cover more extraction scenarios for different pages or to use the generic extractor for scenarios in which that is possible. When all feedback is processed and a pull request is ready to get merged, developers are asked to squash their contribution into a single commit.

Considering the abandoned use of the CI system one could say that the system builds up *technical debt* [2]. That is, developers keep adapting the system through contributions, but over time the functionality added in the past might break. The longer repaying the debt (by restoring the integration of the CI system by fixing defects for successful builds) is postponed, the more time will have to be put into fixing it and the more difficult maintenance becomes. The maintainers might benefit of a proper use of the CI system, as they can process pull requests for which builds fail faster. Though, we should note that the maintainers are avoiding technical debt as well by strictly reviewing the contributions to make sure they adhere to their standards and only quality code is merged into the codebase.

As contributions are made continuously to the project, the maintainers of youtube-dl release a new version of the application by creating GitHub releases. Releases are made at least once a month, but often weekly or even daily and are then made available on the main website. The application is also available as distribution packages, but the versions provided are mostly outdated. Therefore the authors recommend their users to follow the manual installation instructions instead to always have the latest version.

Features

Youtube-dl offers many other features than just downloading a video using a provided URL. All of the features are listed and explained in the `README.md` that is available on the repository. The features can be split into 5 categories, namely:

- Extractor features
- Download features
- Post-processing features
- Configuration files
- Testing

The **extractor features** relate to the process of extracting information from the provided URL. The most important features here are the support for *playlists* and the *workarounds* that are available to support certain websites. If a URL contains a playlist, youtube-dl is able to download the entire playlists. These playlists can be filtered based on different available meta-data such as view count, duration, liked, titles, etc. The workarounds allow the user to specify custom header fields (like a browser) to work around some security measures that might have been taken by the site to avoid downloading the content.

The **download features** are related to the actual download process. Examples of such features are the ability to use a *proxy*, *authentication* and *quality selection*. The latter one allows you to download the file with exactly the quality that you require (if available). Authentication allows users to specify credentials that can be used to work around permission issues. Authentication is also possible by using a user-defined cookie (in the case that regular authentication is not supported or fails). The use of a proxy allows a user to access content that is not available from the users normal connection (think of geo-limitations).

Within the **post-processing features** the user is able to tweak the downloaded file to his/her preference. A user can among others choose to *embed subtitles* and *convert the content*. The user can specify a subtitle source, which then will be embedded in a video file. The user can also specify the desired output format. The post-processor in question will convert the downloaded file into the desired format.

In order to ease the use of youtube-dl it is possible to create a **configuration file**. Normally one would need to specify all options on the command-line. Using this file one can specify options that he/she uses regularly. These options then will be executed automatically by youtube-dl. For example, instead of typing `youtube-dl URL -x --proxy SOME_IP -o ~/Movies/%(title)s.%(ext)s` every time one could create a configuration file like this;

```
-x
--proxy SOME_IP
-o ~/Movies/%(title)s.%(ext)s
```

The only remaining thing one has to do to achieve the same result, is running `youtube-dl URL`.

There are some **testing features** created especially for Developers and Supporters of the project. They consist, among others, out of *error reporting*, *logging* and *simulation*. The latter allows the developers to simulate a download in order to find or trace potential bugs.

All of the features that are available can be ran independent from each other. Some features depend on external sources. An example of such a dependency is present in the automatic retrieval of subtitles from YouTube videos. This requires information from YouTube (that has to be available in order to succeed).

Regulations

Youtube-dl allows to download several types of media from the internet. Following from this possibility youtube-dl allows users to download potentially copyrighted material. Youtube-dl will have to adhere to local laws. In this section we will discuss matters related to such and other relevant regulations.

License

Youtube-dl is released in the public domain. This means that everyone is free to copy, modify, sell, distribute, etc. There are users that have built extensions and redistributed the new package. Some users actually added support for sites that are known to have issues with copyright infringement. But this is independent of the official repository. Since anyone can add functionality one might include support for illegal sources as well. This, however, is not something that can be influenced by the youtube-dl team.

Practice

Youtube-dl receives a lot of site support requests that ask for support of sites that infringe copyrights. The authors of youtube-dl respect the copyright laws; they refuse to add support for sites that clearly are intended to infringe copyright (see for example issue #1048). As long as there are videos on the website that do *not* infringe copyrights, support for that particular website will likely be allowed. This is also written in their `README.md`. Based on this support for sites like YouTube would be allowed, but actually using the application for YouTube would be illegal (breaching their Terms and Conditions).

Laws

The internet allows its users to spread all kinds of materials very easily. Unfortunately this also allows these users to spread illegally obtained materials with the same ease. Consequently, these illegally distributed materials can be downloaded with the use of tools like youtube-dl. Since youtube-dl originally was designed to download from YouTube, we will discuss the case of YouTube in more detail.

YouTube offers video streaming as a service. To monetize this service they show ads on their website (amongst others). In their Terms and Conditions they state the following;

> **Point 5.1 - L** - you agree not to access Content or any reason other than your personal, non-commercial use solely as intended through and permitted by the normal functionality of the Service, and solely for Streaming. "Streaming" means a contemporaneous digital transmission of the material by YouTube via the Internet to a user operated Internet enabled device in such a manner that the data is intended for real-time viewing and not intended to be downloaded (either permanently or temporarily), copied, stored, or redistributed by the user.

It is stated that anything that is not streaming, is not allowed. This would mean that the usage of youtube-dl for downloading is not allowed either. The Terms and Conditions are not the same for every country. The quote above is taken from the UK Terms and Conditions. However in the worldwide version it states;

> **Point 5 - B** - Content is provided to you AS IS. You may access Content for your information and personal use solely as intended through the provided functionality of the Service and as permitted under these Terms of Service. You shall not download any Content unless you see a “download” or similar link displayed by YouTube on the Service for that Content. You shall not copy, reproduce, distribute, transmit, broadcast, display, sell, license, or otherwise exploit any Content for any other purposes without the prior written consent of YouTube or the respective licensors of the Content. YouTube and its licensors reserve all rights not expressly granted in and to the Service and the Content.

Here it is mentioned that it is allowed to download content, but only when they show a download link on YouTube itself. So downloading the content is not allowed in this scenario. However, the regulations vary based on location, this is something that youtube-dl should consider.

Laws are different for every country. For instance the Canadian government published An Act to amen the Copyright Act. It states that you are free to download any content that is legally acquired by you, as long as you do not redistribute. You might, however, still breach the Terms and Conditions of a specific website. If we look at The Netherlands we see a different situation. In The Netherlands it was always allowed to download any content from the web, only distributing was illegal. After a decision in the Court of Justice of the European Union it was said to be illegal in all cases to download copyrighted contents. In Germany there are several complaints about users who were distributing content and who received a fine for that.

Youtube-dl itself can be considered legal. Although it allows one to download copyrighted material, one still has to make the decision to put youtube-dl to that use. Sites can offer both copyrighted and free contents, for the latter one support of youtube-dl is legitimate. As developers refuse to merge contributions that include parts that are not following their rules, youtube-dl will also remain legal in the future. If the user chooses to download illegal content, he/she will be the one responsible for the consequences.

Conclusion

Youtube-dl is a command line tool that allows one to download video content from one of the websites supported by the application. Because of this simple interface and the use of the Python language, all major operating systems are supported. The behavior of the program can be influenced by passing additional options on the command-line. Nowadays there are a lot of options available that can be used to one's preference.

Youtube-dl has a rather simple architecture. It can be divided into four major components; the core of the application, the downloaders, the extractors and the post-processors. All offer different and specific functionality to the application. This well defined structure allows for continuous updates as external contributors are not required to have an extensive amount of prerequisite knowledge about the whole application. This is one of the main reasons why the application can constantly be extended with new valuable functionality.

The simplicity of youtube-dl's architecture allows for an easy, continuous, integration of new functionality. Some of the more active maintainers have been granted collaborator access to the repository. They ensure that all contributions to the project adhere to the strict guidelines. Because of the active community youtube-dl offers a wide range of functionality and an exceptional amount of supported websites.

Besides, youtube-dl offers a way to download both legal and illegal content from supported websites. Apart from the fact whether the content in case is legal or illegal, websites might still disallow downloading content in general in their *Terms of Service*. Youtube-dl also has to deal with the various regulations in each country. Depending on the country, these regulations do not allow downloading copyrighted material. The maintainers tend to respect the regulations and decline to add support for websites that infringe copyright. The developers of youtube-dl are, however, not responsible for the activities of the users and thus the application itself can be considered legal.

Because of the simple architecture, the extensive reviews of contributions and the attitude of the maintainers towards the regulations, we think that youtube-dl deserves its high standing on GitHub.

References

[1] Nick Rozanski and Eoin Woods. Software Systems Architecture: Working with Stakeholders using Viewpoints and Perspectives. Addison-Wesley, 2012.

[2] Chris Cairns and Sarah Allen. 2015. What is technical debt? Retrieved February 28, 2016, from: `https://18f.gsa.gov/2015/09/04/what-is-technical-debt/`