# Modeling the Exception Flow in PHP Systems

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Tom den Braber
born in Maassluis, the Netherlands

Software Engineering Research Group
Department of Software Technology
Faculty of Electrical Engineering, Mathematics & Computer Science, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Moxio BV
Rotterdamseweg 183c
Delft, the Netherlands
www.moxio.com

©2017 Tom den Braber. All rights reserved.

# Modeling the Exception Flow in PHP Systems

Author: Tom den Braber
Student id: 4223780
Email: tomdenbraber@gmail.com

**Abstract**

The goal of this thesis is to learn how exception handling constructs are used by PHP developers. We present an approach for detecting the exception flow of a software system, based on the work of Robillard and Murphy [26]. We show the accuracy of this approach by evaluating the tool on a corpus of three different PHP systems. The approach is thereafter used to perform an empirical study on a corpus of 20 PHP systems. For each of these systems, we compute the exception flow and measure the number of exceptions that are encountered, how often exceptions are propagated before they are caught, by what type they are typically caught, and whether they are documented. The results show that many exceptions are propagated often before they are caught and that many are caught by subsumption. Another finding is that exceptions are often not documented, which in many cases is a violation of the Liskov Substitution Principle.

Thesis Committee:

Chair: Prof. Dr. A. van Deursen
University supervisors: MSc. M. Soltani, Faculty EEMCS, TU Delft
Dr. M. Aniche, Faculty EEMCS, TU Delft
Company supervisor: MSc. M. Wijngaard, Moxio BV
Committee Members: Dr. G. Gousios, Faculty EEMCS, TU Delft
Dr. S. Verwer, Faculty EEMCS, TU Delft

# Preface

After a journey of almost 9 months, I can now deliver a thesis of which I am proud and for which I worked hard. I would like to thank the people who supported me throughout the research process. To start with, I want to thank Mozhan, who kindly reviewed my work and has been a very helpful supervisor. I would also like to thank Merijn for the constructive meetings we had over the course of the project. The third person who deserves my thanks is Mauricio, who pushed me to pursue a high standard concerning the determination of the accuracy of the algorithm. My thanks also go out to Arie, for his feedback and ideas. I want to thank Hylke for giving me the opportunity to perform this study at Moxio. I am grateful for all my colleagues at Moxio, with whom I had a lot of fun. I also want to thank all the people who were willing to work in my house, painting walls, doors and window frames, while I was working on this thesis. Finally, I would like to thank Lisette for supporting me throughout the whole project.

Tom den Braber<br>Delft, the Netherlands<br>August 16, 2017

# Contents

**Preface** iii

**Contents** v

**List of Figures** vii

**1 Introduction** 1

**2 Background** 3
- 2.1 The PHP language . . . 3
- 2.2 Static Analysis to Support the Evolution of Exception Structure in Object-Oriented Systems . . . 4
- 2.3 Simple and Efficient Construction of Static Single Assignment Form . . . 6
- 2.4 Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis . . . 7

**3 Approach** 9
- 3.1 Type Inference . . . 10
- 3.2 Call Graph Construction . . . 13
- 3.3 Flow Detection . . . 16
- 3.4 Research Scope . . . 18

**4 Algorithm Accuracy** 21
- 4.1 Methodology . . . 21
- 4.2 Results . . . 25
- 4.3 Concluding remarks . . . 30

**5 Empirical Study** 33
- 5.1 Research Questions . . . 33
- 5.2 Corpus . . . 34
- 5.3 Methodology . . . 35

5.4 Threats to Validity . . . . . . . . . . 38

**6 Results** **39**
6.1 **RQ1.** Number of Encounters per Method . . . . . . . . . . 39
6.2 **RQ2.** Paths of Exceptions . . . . . . . . . . 39
6.3 **RQ3.** Catch by Subsumption . . . . . . . . . . 39
6.4 **RQ4.** Documentation Validation . . . . . . . . . . 41

**7 Discussion** **45**
7.1 **RQ1.** Number of Encounters per Method . . . . . . . . . . 45
7.2 **RQ2.** Paths of Exceptions . . . . . . . . . . 45
7.3 **RQ3.** Catch by Subsumption . . . . . . . . . . 46
7.4 **RQ4.** Documentation Validation . . . . . . . . . . 47
7.5 Algorithm Accuracy . . . . . . . . . . 48

**8 Related Work** **49**
8.1 Exception Flow Modeling . . . . . . . . . . 49
8.2 Exception and Error Handling . . . . . . . . . . 49
8.3 PHP and Web Development . . . . . . . . . . 50

**9 Conclusion** **51**
9.1 Contributions . . . . . . . . . . 51
9.2 Conclusions . . . . . . . . . . 51
9.3 Future Work . . . . . . . . . . 52

**Bibliography** **53**

**A Results** **57**
A.1 Paths of Exceptions . . . . . . . . . . 57
A.2 Catch by Subsumption . . . . . . . . . . 61
A.3 Documentation Validation . . . . . . . . . . 71

# List of Figures


2.1 Catch clauses in PHP . . . . . . 4
2.2 Example code for scopes and guarded scopes . . . . . . 5
2.3 Indication of the (guarded) scopes in code in Listing 2.2. . . . . . . 5
2.4 Example class hierarchy . . . . . . 8

3.1 Overview of the approach for computing the exception flow . . . . . . 10
3.2 Exception flow example code . . . . . . 13
3.3 Visual representation of the exception flow of the program in Listing 3.2. . . . 13
3.4 Dynamic calls . . . . . . 19

4.1 Real errors and propagated errors . . . . . . 22
4.2 Example situation for exception entering scope via different routes . . . . . . 24
4.3 Type inference limitations . . . . . . 30

5.1 A method with a `@throws` annotation . . . . . . 34
5.2 An example where the exception can be propagated from two sources . . . . . 36
5.3 Propagation via multiple paths for exception *e* to the catch clause in scope c . . 37
5.4 Catch by subsumption code example . . . . . . 37
5.5 Catch by subsumption example class hierarchy . . . . . . 37

6.1 Shortest path lengths from `throw` statement to `catch` clause . . . . . . 40
6.2 Distances between exceptions, catch clauses and root types in all projects . . . 41
6.3 Distances between exceptions, catch clauses and root types in PHPUnit . . . . 42
6.4 Comparison of `throws` annoations to encountered exceptions . . . . . . 43

A.1 Shortest path lengths from `throw` statement to `catch` clause in Joomla . . . . . 57
A.2 Shortest path lengths from `throw` statement to `catch` clause in CakePHP . . . 57
A.3 Shortest path lengths from `throw` statement to `catch` clause in Doctrine . . . . 58
A.4 Shortest path lengths from `throw` statement to `catch` clause in DokuWiki . . . 58
A.5 Shortest path lengths from `throw` statement to `catch` clause in Drupal . . . . . 58
A.6 Shortest path lengths from `throw` statement to `catch` clause in Fabric . . . . . 58

A.7 Shortest path lengths from `throw` statement to `catch` clause in Nette . . . . . . 59
A.8 Shortest path lengths from `throw` statement to `catch` clause in Phing . . . . . . 59
A.9 Shortest path lengths from `throw` statement to `catch` clause in phpBB . . . . . 59
A.10 Shortest path lengths from `throw` statement to `catch` clause in phpDocumentor2 59
A.11 Shortest path lengths from `throw` statement to `catch` clause in PHPUnit . . . . 60
A.12 Shortest path lengths from `throw` statement to `catch` clause in Digitaalloket . . 60
A.13 Shortest path lengths from `throw` statement to `catch` clause in Objectbrowser . 60
A.14 Shortest path lengths from `throw` statement to `catch` clause in Roundcube . . . 60
A.15 Shortest path lengths from `throw` statement to `catch` clause in Smarty . . . . . 61
A.16 Shortest path lengths from `throw` statement to `catch` clause in Symfony . . . . 61
A.17 Shortest path lengths from `throw` statement to `catch` clause in Wordpress . . . 61
A.18 Shortest path lengths from `throw` statement to `catch` clause in Zend Framework 61
A.19 Distances from all caught exceptions to the types of the catching catch clause in Joomla . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 62
A.20 Distances from all caught exceptions to the types of the catching catch clause in CakePHP . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 62
A.21 Distances from all caught exceptions to the types of the catching catch clause in Doctrine . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 62
A.22 Distances from all caught exceptions to the types of the catching catch clause in DokuWiki . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 62
A.23 Distances from all caught exceptions to the types of the catching catch clause in Drupal . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 63
A.24 Distances from all caught exceptions to the types of the catching catch clause in Fabric . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 63
A.25 Distances from all caught exceptions to the types of the catching catch clause in Nette . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 63
A.26 Distances from all caught exceptions to the types of the catching catch clause in Phing . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 63
A.27 Distances from all caught exceptions to the types of the catching catch clause in phpBB . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 64
A.28 Distances from all caught exceptions to the types of the catching catch clause in phpDocumentor2 . . . . . . . . . . . . . . . . . . . . . . . . . . . 64
A.29 Distances from all caught exceptions to the types of the catching catch clause in PHPUnit . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 64
A.30 Distances from all caught exceptions to the types of the catching catch clause in Digitaalloket . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 64
A.31 Distances from all caught exceptions to the types of the catching catch clause in Objectbrowser . . . . . . . . . . . . . . . . . . . . . . . . . . . . 65
A.32 Distances from all caught exceptions to the types of the catching catch clause in Roundcube . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 65
A.33 Distances from all caught exceptions to the types of the catching catch clause in Smarty . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 65
A.34 Distances from all caught exceptions to the types of the catching catch clause in Symfony . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 65

A.35 Distances from all caught exceptions to the types of the catching catch clause in Wordpress . . . . . 66
A.36 Distances from all caught exceptions to the types of the catching catch clause in Zend Framework . . . . . 66
A.37 Distance from the caught exceptions to the `Exception` type in Joomla . . . . . 67
A.38 Distance from the caught exceptions to the `Exception` type in CakePHP . . . . 67
A.39 Distance from the caught exceptions to the `Exception` type in Doctrine . . . . 67
A.40 Distance from the caught exceptions to the `Exception` type in DokuWiki . . . 67
A.41 Distance from the caught exceptions to the `Exception` type in Drupal . . . . . 68
A.42 Distance from the caught exceptions to the `Exception` type in Fabric . . . . . 68
A.43 Distance from the caught exceptions to the `Exception` type in Nette . . . . . . 68
A.44 Distance from the caught exceptions to the `Exception` type in Phing . . . . . . 68
A.45 Distance from the caught exceptions to the `Exception` type in phpBB . . . . . 69
A.46 Distance from the caught exceptions to the `Exception` type in phpDocumentor2 69
A.47 Distance from the caught exceptions to the `Exception` type in PHPUnit . . . . 69
A.48 Distance from the caught exceptions to the `Exception` type in Digitaalloket . . 69
A.49 Distance from the caught exceptions to the `Exception` type in Objectbrowser . 70
A.50 Distance from the caught exceptions to the `Exception` type in Roundcube . . . 70
A.51 Distance from the caught exceptions to the `Exception` type in Smarty . . . . . 70
A.52 Distance from the caught exceptions to the `Exception` type in Symfony . . . . 70
A.53 Distance from the caught exceptions to the `Exception` type in Wordpress . . . 71
A.54 Distance from the caught exceptions to the `Exception` type in Zend Framework 71
A.55 Annotations compared to explicitly raised exceptions in CodeIgniter . . . . . . 72
A.56 Annotations compared to explicitly raised exceptions in Joomla . . . . . . . . . 72
A.57 Annotations compared to explicitly raised exceptions in PEAR . . . . . . . . . 72
A.58 Annotations compared to explicitly raised exceptions in CakePHP . . . . . . . 72
A.59 Annotations compared to explicitly raised exceptions in Doctrine . . . . . . . . 73
A.60 Annotations compared to explicitly raised exceptions in DokuWiki . . . . . . . 73
A.61 Annotations compared to explicitly raised exceptions in Drupal . . . . . . . . . 73
A.62 Annotations compared to explicitly raised exceptions in Fabric . . . . . . . . . 73
A.63 Annotations compared to explicitly raised exceptions in Nette . . . . . . . . . 74
A.64 Annotations compared to explicitly raised exceptions in Phing . . . . . . . . . 74
A.65 Annotations compared to explicitly raised exceptions in phpBB . . . . . . . . . 74
A.66 Annotations compared to explicitly raised exceptions in phpDocumentor2 . . . 74
A.67 Annotations compared to explicitly raised exceptions in PHPUnit . . . . . . . . 75
A.68 Annotations compared to explicitly raised exceptions in Digitaalloket . . . . . 75
A.69 Annotations compared to explicitly raised exceptions in Objectbrowser . . . . . 75
A.70 Annotations compared to explicitly raised exceptions in Roundcube . . . . . . 75
A.71 Annotations compared to explicitly raised exceptions in Smarty . . . . . . . . . 76
A.72 Annotations compared to explicitly raised exceptions in Symfony . . . . . . . 76
A.73 Annotations compared to explicitly raised exceptions in Wordpress . . . . . . . 76
A.74 Annotations compared to explicitly raised exceptions in Zend Framework . . . 76
A.75 Annotations compared to encountered (but not raised) exceptions in CodeIgniter 77
A.76 Annotations compared to encountered (but not raised) exceptions in Joomla . . 77

A.77 Annotations compared to encountered (but not raised) exceptions in PEAR . . . 77
A.78 Annotations compared to encountered (but not raised) exceptions in CakePHP . 77
A.79 Annotations compared to encountered (but not raised) exceptions in Doctrine . 78
A.80 Annotations compared to encountered (but not raised) exceptions in DokuWiki 78
A.81 Annotations compared to encountered (but not raised) exceptions in Drupal . . 78
A.82 Annotations compared to encountered (but not raised) exceptions in Fabric . . . 78
A.83 Annotations compared to encountered (but not raised) exceptions in Nette . . . 79
A.84 Annotations compared to encountered (but not raised) exceptions in Phing . . . 79
A.85 Annotations compared to encountered (but not raised) exceptions in phpBB . . 79
A.86 Annotations compared to encountered (but not raised) exceptions in phpDocumentor2 . . . 79
A.87 Annotations compared to encountered (but not raised) exceptions in PHPUnit . 80
A.88 Annotations compared to encountered (but not raised) exceptions in Digitaalloket 80
A.89 Annotations compared to encountered (but not raised) exceptions in Object-browser . . . 80
A.90 Annotations compared to encountered (but not raised) exceptions in Roundcube 80
A.91 Annotations compared to encountered (but not raised) exceptions in Smarty . . 81
A.92 Annotations compared to encountered (but not raised) exceptions in Symfony . 81
A.93 Annotations compared to encountered (but not raised) exceptions in Wordpress 81
A.94 Annotations compared to encountered (but not raised) exceptions in Zend Framework . . . 81
A.95 Annotations at abstract method level compared to encountered exceptions in implementing methods in Joomla . . . 82
A.96 Annotations at abstract method level compared to encountered exceptions in implementing methods in CakePHP . . . 82
A.97 Annotations at abstract method level compared to encountered exceptions in implementing methods in Doctrine . . . 82
A.98 Annotations at abstract method level compared to encountered exceptions in implementing methods in DokuWiki . . . 82
A.99 Annotations at abstract method level compared to encountered exceptions in implementing methods in Drupal . . . 83
A.100 Annotations at abstract method level compared to encountered exceptions in implementing methods in Fabric . . . 83
A.101 Annotations at abstract method level compared to encountered exceptions in implementing methods in Nette . . . 83
A.102 Annotations at abstract method level compared to encountered exceptions in implementing methods in Phing . . . 83
A.103 Annotations at abstract method level compared to encountered exceptions in implementing methods in phpBB . . . 84
A.104 Annotations at abstract method level compared to encountered exceptions in implementing methods in phpDocumentor2 . . . 84
A.105 Annotations at abstract method level compared to encountered exceptions in implementing methods in PHPUnit . . . 84

A.106 Annotations at abstract method level compared to encountered exceptions in implementing methods in Digitaalloket . . . . . . . . . . . . . . . . . . . . . . . 84
A.107 Annotations at abstract method level compared to encountered exceptions in implementing methods in Objectbrowser . . . . . . . . . . . . . . . . . . . . . . 85
A.108 Annotations at abstract method level compared to encountered exceptions in implementing methods in Roundcube . . . . . . . . . . . . . . . . . . . . . . . 85
A.109 Annotations at abstract method level compared to encountered exceptions in implementing methods in Smarty . . . . . . . . . . . . . . . . . . . . . . . . . . 85
A.110 Annotations at abstract method level compared to encountered exceptions in implementing methods in Symfony . . . . . . . . . . . . . . . . . . . . . . . . . 85
A.111 Annotations at abstract method level compared to encountered exceptions in implementing methods in Wordpress . . . . . . . . . . . . . . . . . . . . . . . . 86
A.112 Annotations at abstract method level compared to encountered exceptions in implementing methods in Zend Framework . . . . . . . . . . . . . . . . . . . . 86

# Chapter 1

# Introduction

Exception handling mechanisms, introduced by Goodenough [19], provide a means to signal an exceptional situation and subsequently handle that situation. Different exception types are used to signal different exceptional situations, and for each exception type a block of code can be created to handle that specific exception. Exception handling mechanisms enable developers to separate the normal flow of the program from the exceptional flow.

These mechanisms are often misused and misunderstood. For example, Cabral and Marques [10] found that exceptions are often not used as an error recovery system. Instead of handling the exceptional situation, the exception handling constructs are used for logging the error, notifiying the user or terminating the application. Another aspect of exception handling mechanisms is that classes that use them are more often defect prone than classes that do not use them [23]. Miller and Tripathi [24] argue that the current mechanisms used for exception handling are flawed. For instance, they claim that exception handling mechanisms place an extra burden on a developer as they have to understand two different execution paths: the exception flow path and the normal execution path. When combining the conclusions of Cabral and Marques [10], Marinescu [23] and Miller and Tripathi [24], we see that exception handling mechanisms are hard to understand.

Because reasoning about the exception flow is hard, several static analysis tools for inspecting the exception flow of a software system have been developed (e.g. [26, 11, 18]). However, these tools are all developed for languages and environments like Java and .NET. To the best of our knowledge, no such tool exists for PHP.

In this thesis, we present a static analysis tool for deducing the exception flow of PHP programs, based on the work of Robillard and Murphy [26]. This approach works by analysing the AST of the program. We consider only the structured programming features of PHP. This means that we account for the exception flow in methods and functions, but that we do not focus on PHP code embedded in HTML code. The focus of our thesis is on the exception flow as created by the programmer; we disregard the exceptions that are generated by the PHP environment.

We evaluated the accuracy of the approach by creating oracles of three PHP software systems, which consists of two open-source projects and one closed-source project. The evaluation shows that 63% of the exceptions that are encountered in a system are actually also detected by the presented algorithm.

We thereafter conducted an empirical study on 20 software projects, totalling over 1.6 million lines of PHP code. We performed this study using our static analysis tool. The empirical study shows that about 30% of the caught exceptions is not caught close to their source. We also found that many exceptions are not documented. We argue that missing annotations in interfaces are a violation of the Liskov Substitution Principle and show that this principle is violated often in PHP systems.

The remainder of this thesis is organised as follows. In Chapter 2, we introduce the PHP language, as well as the algorithms on which our approach is based. Chapter 3 describes the algorithms that we devised to deduce the exception flow of a PHP program. Chapter 4 contains an analysis of the accuracy of the algorithm. In Chapter 5, we detail the empirical study that was conducted on a corpus of 20 software projects. Chapter 6 shows the results of this empirical study. We discuss and interpret the results of the empirical study in Chapter 7. In Chapter 8, we show how our work relates to other research done on exception flow, exception handling, and PHP. Finally, Chapter 9 concludes the thesis.

# Chapter 2

# Background

As our research focuses on PHP, we first introduce the PHP language and its exception handling constructs in Section 2.1. Our research is to a large extent based on existing approaches, proposed in literature. In this chapter, we present these approaches. Section 2.2 presents the work by Robillard and Murphy [26], which concerns deducing the exception flow of software programs. Thereafter, we present the work of Braun et al. [9], which introduces an algorithm for creating a representation in Static Single Assignment (SSA) form of a program. Section 2.4 presents the work of Dean et al. [13] concercing Class Hierarchy Analysis.

## 2.1 The PHP language

PHP is a programming language focused on server-side application development. It currently ranks 7th on the TIOBE programming community index, which indicates its popularity [6]. PHP is dynamically typed, meaning that types of expressions are determined at run-time. The language contains a single-inheritance class model, including interfaces and traits. It also contains a number of dynamic features, e.g. a variable can be used to access methods, functions or other variables.

PHP contains several constructs for dealing with exceptions. To start with, the `throw` keyword can be used to throw an exception. Only objects that implement the `Throwable` interface[1] can be thrown. PHP defines several types of exceptions which can be used and extended, e.g. `RuntimeException` and `LogicException`. Exceptions can be caught by using the `try-catch-finally` construct. The exceptions that are encountered within the `try` block can be caught by one of the `catch` clauses that is attached to the `try` block.

It is important to note that in PHP, `catch` clauses can influence each other if they are attached to the same `try` block. Consider the code in Listing 2.1. The `try` block encounters an exception of type `RuntimeException`. This exception is caught by the first catch clause. However, the code located in this clause throws a new exception of type `Exception`. Whereas in other popular languages with exception handling constructs, like Java, this last

[1]This interface was introduced in PHP 7.0; in earlier versions, the `Exception` class was the basetype of the exception class hierarchy.

exception would not be caught by this `try-catch` construct, in PHP it is caught by the second catch clause.

It is also possible to attach a `finally` block to a `try` block. The code in the `finally` block will always be executed after the code within the `try` block (and potentially any of the `catch` clauses).

```
1  <?php
2  try {
3    throw new RuntimeException();
4  } catch (RuntimeException $e) {
5    throw new Exception();
6  } catch (Exception $e) {
7    log($e);
8  } finally {
9    print "Will always be executed";
10 }
```

Listing 2.1: Catch clauses in PHP

## 2.2 Static Analysis to Support the Evolution of Exception Structure in Object-Oriented Systems

Robillard and Murphy [26] present an approach for deducing the exception flow which is "in theory applicable to any object-oriented programming language that define exceptions as objects." As this criterion also applies to PHP, their approach forms the basis of our research. In Section 2.2.1 we introduce the terminology used by Robillard and Murphy [26]. Section 2.2.2 presents the mathematical definitions of the concepts that they use for computing the exception flow.

### 2.2.1 Terminology

To start with, Robillard and Murphy [26] introduce *scopes* and *guarded scopes*. A scope $s = (I, G)$ consists of a set of instructions $I$ and a set of guarded scopes $G$ which are nested in scope $s$. An exception that is encountered within a scope always flows directly to the boundary of that scope. This means that a scope represents one 'step' that an exception takes in the program as it propagates from one scope to another. A guarded scope $g = (n, C)$ consists of a nested scope $n$ and a sequence $C$ of catch clauses. The catch clauses can prevent exceptions that are encountered in $n$ to flow to the scope that encloses the guarded scope $g$. Each catch clause can catch a certain type of exception, as well as all of its subtypes. If an exception is not caught by its actual type, but by the type of one of its supertypes, it is caught 'by subsumption'.

An example of scopes and guarded scopes can be found in Listing 2.2. This example shows the mapping from the scope definitions of Robillard and Murphy [26] to scopes in

PHP. The scope $s_1$ corresponds to the function a. It contains the guarded scope $gs_1$, which consists of the entire try-catch construct. The guarded scope itself contains a scope, called $s_2$. $s_2$ consists of all the instructions in the try block of $gs_1$.

```
1 <?php
2 function a() {
3   try {
4     throw new SomeException();
5   } catch (SomeException $e) {
6     $this->logger->logException($e);
7   }
8 }
```

Listing 2.2: Example code for scopes and guarded scopes

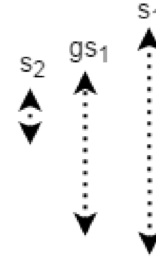



Figure 2.3: Indication of the (guarded) scopes in code in Listing 2.2.

Each scope $s = (I, G)$ can *encounter* exceptions. Four different origins for an exception can be discerned, as the following [26]:

**raises** the set of exceptions that are explicitly thrown in scope $s$ using the throw statement;

**uncaught** the set of exceptions that are encountered in one of the nested guarded scopes $g \in G$ of scope $s$, but that are not caught by one of $g$'s catch clauses;

**propagates** the set of exceptions that are encountered in a scope $t$ where $t$ is not enclosed by $s$ for which holds that $s$ contains a call to scope $t$;

**generates** the set of exceptions for which holds that they are generated by an instruction $i \in I$ as the result of a system operation in the instruction $i$, where $i$ is not a throw statement.

When we compute the union of these four different sets, we have the complete set of exceptions that can be encountered in a scope $s$. This set is called the *encounters* set of scope $s$.

### 2.2.2 Algorithm

The terms presented in Section 2.2.1 can be defined in mathematical terms. We use the mathematical definitions of these terms in the description of our algorithms. To start with, we define two helper functions. Thereafter, we present a mathematical function for each of the origins presented in Section 2.2.1.

The algorithm uses two helper functions: $encounters(s)$ and $catches(g, c)$. Both these functions return a set of exceptions. The *encounters* function represents the complete set of exceptions encountered in a method; this means that it is defined as the union of the *raises*, *uncaught*, *propagates* and *generates* sets. The input for the *encounters* function is a scope $s = (I, G)$.

$$encounters(s) = raises(I) \cup generates(I) \cup propagates(I) \cup uncaughtGS(G) \tag{2.1}$$

The second helper function, $catches(g,c)$ operates on a guarded scope $g = (s,C)$ where $C = \{c_1, \ldots c_n\}$. $type(c_i)$ denotes the type that catch clause $c_i$ can catch, and $<:$ denotes the subtype relation. Using these notations, the *catches* function is defined as:

$$catches(g,c_i) = \{e | e \in encounters(s) \land e \notin \bigcup_{j=1..i-1} catches(g,c_j) \land e <: type(c_i)) \quad (2.2)$$

Using the *catches* function, the *catchesGS* function is defined as the set of exceptions that are caught by a guarded scope:

$$catchesGS(g,C) = \bigcup_{\forall c \in C} catches(g,c) \quad (2.3)$$

The *encounters* function uses all the sets that were presented in Section 2.2.1. Each of these sets can be formally defined as well. Note that there is a difference between the *uncaught* set and the other sets: the *uncaught* set is calculated using a set of guarded scopes $G$, whereas the other sets are calculated over a set of instructions $I$. We present the definition of the *raises* set here, but the *generates* and *propagates* sets can be similarly defined.

$$raises(I) = \bigcup_{\forall i \in I} raises(i) \quad (2.4)$$

The $raises(i)$ function returns the set of exceptions that are explicitly raised by instruction $i$. In PHP, this would be when instruction $i$ is a `throw` statement. The $generates(i)$ function returns the set of exceptions that are triggered by internal PHP mechanisms, but not by a `throw` statement. The $propagates(i)$ function returns the set of exceptions that are propagated by instruction $i$. Let $i$ be a method call that can be resolved to a set of scopes $S_m$. The collection of exceptions that can be progated by $i$ is then defined as:

$$propagates(i) = \bigcup_{s \in S_m} encounters(s) \quad (2.5)$$

On a set of guarded scopes $G$, the *uncaughtGS* defines the set of exceptions that are not caught by these guarded scopes.

$$uncaughtGS(G) = \bigcup_{\forall g \in G} uncaught(g) \quad (2.6)$$

On a single guarded scope, the function returns the set of exceptions that is encountered in $g = (s,C)$, but not caught by any of the catch clauses $c \in C$.

$$uncaught(g) = encounters(s) - catchesGS(C) \quad (2.7)$$

## 2.3 Simple and Efficient Construction of Static Single Assignment Form

Braun et al. [9] introduce an algorithm for creating intermediate representations of programs in the Static Single Assignment (SSA) form. In the SSA form, each variable is assigned exactly once. A representation in SSA is useful for performing dependency related analyses,

as it is a compact representation of the *use-def* chains in a program. The algorithm assumes that the program under analysis can be divided into basic blocks, where each block corresponds to a set of linearly executed set of instructions. These blocks can then be connected to form the Control Flow Graph (CFG). For each read of a variable in a basic block, the algorithm tries to relate it to a preceding definition of that variable within the current block. If this is not possible, it tries to resolve it to one of the preceding blocks recursively. If a variable is defined along multiple paths to the current basic block, a $\phi$ function is introduced in the current block, which contains all possible definitions that might reach the current basic block. The approach by Braun et al. [9] is proven to construct the minimal SSA form of a program, i.e. the number of $\phi$ functions is minimal.

This algorithm is used in our research for building CFGs of programs. These CFGs are intermediates, and function as input for the type inference algorithm.

## 2.4 Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis

The work of Robillard and Murphy [26] uses the work of Dean et al. [13] for resolving calls, which is needed to calculate the *propagates* set. The approach that Dean et al. [13] define is called Class Hierarchy Analysis (CHA), as information from the class hierarchy of the program to be analysed is used to statically limit the amount of methods that a call can be resolved to. This approach does not construct a call graph, but it provides the ingredients for creating a call graph. We first present an example of what the effect of CHA is in practice. Thereafter, we briefly present the building blocks of CHA.

To start with, we examine the class hierarchy in Figure 2.4. Consider the situation where method `C.p` contains a call `this.m()`. If `p` is called on an object of type `C`, we can see that the call `this.m()` automatically resolves to `B.m`, as this is the most direct ancestor that implements method `m`. If the method `p` is called on a an object of type `D` or `E`, it resolves to method `C.p`. Within the method `C.p`, the expression `this` has the type `D` respectively `E`, which both also do not implement the method `m`. This means that the call `this.m` also resolves to the method `B.m` for both subclasses of `C`. In summary, the call `this.m()` in method `C.p` always resolves to method `B.m`. This line of reasoning is the foundation of the CHA approach.

Dean et al. [13] introduce two sets: the *applies-to* set and the $\text{Cone}(C)$ set. The applies-to set for a method is the set of classes for which that method is the appropriate target. If we consider the class hierarchy in Figure 2.4, the applies-to set for method `B.m` would be the set of classes $\{B,C,D,E\}$, as for each of these classes, a call to method `m` would resolve to method `B.m`. $\text{Cone}(C)$ is defined as the set of all subclasses of class $C$, including $C$ itself. For the hierarchy in Figure 2.4, $\text{Cone}(B)$ would be the set $\{B,C,D,E\}$.

To compute the applies-to sets, we first compute a partial order of all methods in the system. Within the partial order, a method $m_1$ is smaller than a method $m_2$ iff method $m_1$ overrides $m_2$. After constructing the partial order, for each method $m$ defined on a class $C$ an applies-to set is initialised to $\text{Cone}(C)$. Thereafter, the partial order is traversed top-down. For each visited method $C.m$, each of the direct children are visited and their (initial)

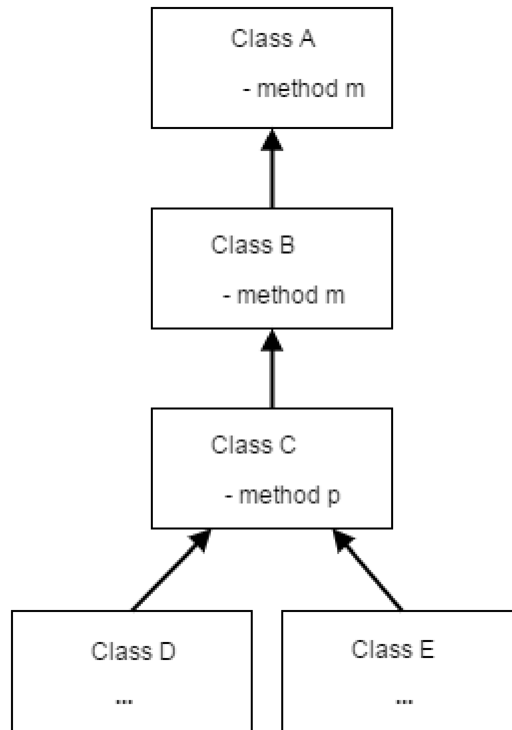



Figure 2.4: Example class hierarchy

applies-to sets are substracted from the applies-to set of $C.m$. If $C.m$ is defined in type $C$ and $D_1, D_2, \ldots, D_n$ are the classes that directly override $C.m$, the applies-to set for $C.m$ can be calculated as follows:

$$\text{Cone}(C) - \{\text{Cone}(D_1) \cup \text{Cone}(D_2) \cup \ldots \cup \text{Cone}(D_n)\} \tag{2.8}$$

Part of our approach consists of creating the call graph, so that the *propagates* sets can be computed. In our research, we use Class Hierarchy Analysis as a basis for constructing the call graph of a program.

# Chapter 3

# Approach

In this chapter, we present our approach for deducing the exception flow in PHP programs. To map the the approach of Robillard and Murphy [26] to the PHP language, we have to overcome two main difficulties. The first difficulty is that the Abstract Syntax Tree (AST) of a PHP program does not contain types. The solution is to first infer the types of a program before starting the detection of the exception flow. The second difficulty lies in the fact that PHP supports traits, which were not included in the original analysis of Robillard and Murphy [26]. A trait is a set of related methods and/or properties, which can be imported into classes independent of their class hierarchy[1].

The approach is visualised in Figure 3.1. Each of the blocks represents a certain action that needs to be taken to get to the end result: the exception flow. The approach that Robillard and Murphy [26] propose operates on the AST (see Section 2.2). To obtain the AST of a program, the code of the program needs to be parsed. Unlike languages with static typing, the AST of a PHP program does not contain any type information. To be able to resolve method calls and deduce the types of the exceptions that are thrown in the program, the types of the expressions in the AST have to be inferred. However, type inference cannot be performed directly on the AST: it needs control flow information to know which expressions can influence each other. To get the information about the control flow, the AST is transformed into a Control Flow Graph (CFG), which can be seen in step 2a in Figure 3.1. We use the algorithm of Braun et al. [9] to create the CFG (see Section2.3). The types are consecutively inferred using the CFG, which can be seen in step 2b. After the types are inferred, a call graph can be constructed. This call graph is built using the approach that is introduced by Dean et al. [13] (see Section 2.4). Using the type information and the call graph, the exception flow can be inferred in step 4.

In the following sections, we explain each of the relevant building blocks in more detail. To start with, we introduce the type inference algorithm in Section 3.1. Section 3.2 presents the construction of the call graph. In Section 3.3, we elaborate on the computation of the exception flow. The chapter closes with a definition of the scope of this research.

[1] In other languages, like Scala and Ruby, this construct is known as a ‘mixin’.

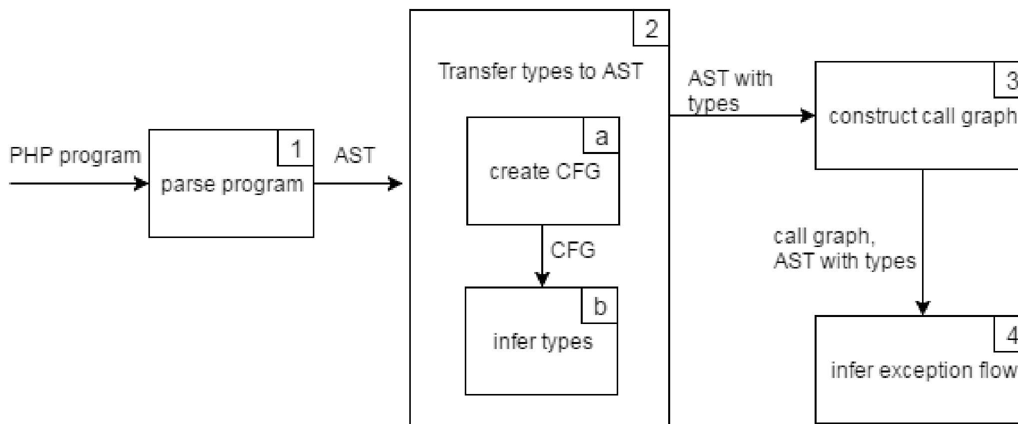



Figure 3.1: Overview of the approach for computing the exception flow

## 3.1 Type Inference

The analysis of Robillard and Murphy [26] operates on the AST of a program. In their work, they assume that the expressions in the AST are typed. However, as a PHP program is dynamically typed, the types are not available when parsing the program: they are determined at run-time. Because the types are needed for resolving method calls and detecting which exceptions are thrown, the first main task is to infer types. We investigated several options for inferring types:

- We examined the work of Van der Hoek and Hage [28]. However, it was not documented, which rendered it unfit for use.

- We also explored the work of Kneuss et al. [21], but their application was last updated five years ago. A considerable amount of work would have to be done to make it run on code that was written for more recent versions of PHP.

- We also considered the PHP-Types[2] tool. Although it was not documented very well, it proved to be easy to use and worked on the latest versions of PHP. PHP-Types yielded acceptable results in an earlier study by Wijngaard [29].

Given these considerations, we decided to use the PHP-Types tool for type inference.

It is not possible to infer types on an AST, as information about the flow of a program is needed for type inference; this flow information is not present in an AST. Therefore, the AST has to be converted to a CFG first. This conversion is done by PHP-CFG[3], which works well together with PHP-Types. PHP-CFG is a PHP implementation of the algorithm by Braun et al. [9], which has been described in more detail in Section 2.3. The CFGs that PHP-CFG creates are intra-procedural. This implies that the program is represented

[2]Source can be found at `https://github.com/ircmaxell/php-types`
[3]Source can be found at `https://github.com/ircmaxell/php-cfg`

as a collection of CFGs, which are not linked together. All code that is not enclosed by a function or method is put into a superficially created *main* function.

When the CFGs have been created, PHP-Types can reconstruct the types of the variables in the CFGs to a certain extent. Not all types can be reconstructed, e.g. due to dynamically accessed variables or the absence of type hints. Algorithm 1 presents a global overview of the algorithm of PHP-Types. The input of this algorithm is an "environment" object, which contains all the declarations and expressions of a PHP program. The environment is created by traversing over all the CFGs of that program and gathering all relevant statements. The notion of *iterations* in the algorithm is introduced because reconstructing the type of one expression might enable the algorithm to reconstruct the type of another expression. In each iteration, the algorithm tries to resolve as many expressions as possible. The algorithm reaches a fix-point if after an iteration no new expressions have been resolved.

**Algorithm 1** Type inference algorithm

**Input:** An *environment* of a program, which comprises all declarations and expressions of the program
**Output:** A list of resolved and unresolved expressions respectively

```
 1: function RESOLVE(environment)
 2:     RESOLVEPROPERTIES(GETPROPERTIES(environment))
 3:     unresolved ← ∅
 4:     resolved ← ∅
 5:     for all exp ∈ GETEXPRESSIONS(environment) do
 6:         if ISLITERAL(exp) then
 7:             resolved[exp] ← RESOLVEFROMVALUE(exp)
 8:         else if exp = $this then    ▷ $this refers to the current instance of the type it
    resides inalg
 9:             resolved[exp] ← RESOLVEFROMDECLARATION(exp)
10:         else
11:             unresolved[exp] ← TYPE(unknown)
12:         end if
13:     end for
14:     repeat
15:         for all exp ∈ unresolved do
16:             type ← RESOLVEVAR(exp)
17:             if ISRESOLVED(exp) then
18:                 resolved[exp] ← type
19:                 unresolved ← unresolved − {exp}
20:             end if
21:         end for
22:     until ¬EMPTY(unresolved) and NUMBEROFEXPRRESOLVEDINLASTITERATION
    = 0
23:     return resolved, unresolved
24: end function
```

The algorithm uses four resolution functions, which have the following functionality:

- The `resolveProperties` function loops over all properties and tries to find a corresponding phpdoc[4] annotation. If an annotation can be found, the type from that annotation is taken to be the type of the property.
- The `resolveFromValue` function takes a literal and infers the type from the value of the literal. For example, it infers the type 'string' from a literal with the value `'abc'`.
- The `resolveFromDeclaration` function resolves the types from '`$this` expressions'. In PHP, the `$this` keyword refers to the current instance of the type the `$this` expression resides in. The type of `$this` can thus be inferred to the type in which the expression resides.
- `resolveVar` applies a variety of techniques to reconstruct the type of an expression:
  - Type hints are parsed (e.g. typed parameters);
  - Documented type annotations are parsed (i.e. @return, @param and @var declarations in phpdoc);
  - Types are reconstructed for PHP expressions with a 'default' type (e.g. `($a === $b)` has the type 'boolean');
  - An internal database containing the return types of all PHP functions is used to assign types to expressions containing a call to such a function;
  - Types of method call expressions are reconstructed by resolving method calls to the method defined in the type of the callee object or in any of its subtypes. Calls are resolved without considering the context of the call, which means that the resolution algorithm is context-insensitive.

The original version of PHP-Types did not support traits: it did not take them into account when building the type hierarchy of a system. A trait is a set of related methods, which can be imported into several independent classes. A trait can be seen as a normal interface, with the only difference being that methods defined in a trait can carry an implementation. Because our industry partner uses of traits in several of its projects, we decided to extend the type inference system to also contain support for traits[5].

When running PHP-Types on the created CFGs, the types are inferred onto the CFGs, but not onto the AST. The representation that PHP-CFG uses differs from the representation as generated by the parser, which implies that there is no direct link between the CFGs and the AST. To be able to map the types from the CFGs back to the AST, the PHP-CFG tool was adapted[6]. When the PHP-CFG tool creates a new CFG-node based on an existing AST-node, the CFG-node now 'remembers' from which AST-node it originated. This information is then used to map the types from expressions in the CFG back to the AST.

[4] phpdoc is a documentation standard, see https://www.phpdoc.org/

[5] The adapted version can be found at `https://github.com/tomdenbraber/php-types`

[6] The adapted version can be found at `https://github.com/tomdenbraber/php-cfg`

## 3.2 Call Graph Construction

To be able to detect the exception flow, the call graph of a program has to be computed. The exception flow can be seen as a 'reverse flow': it has the reverse direction of the call graph, in the sense that a call goes from a caller to a callee and an exception is propagated from a callee to a caller. This is shown in the code in Listing 3.2 and the corresponding exception flow graph in Figure 3.3.

Robillard and Murphy [26] use the work of Dean et al. [13] for building the call graph (see Section 2.4), which we also use in our work. In this section, we describe our implementation of their algorithm. We first explain how we fill the partial order with methods. Thereafter, we explain how we use the partial order to create a *call resolution map*. This call resolution map is a different representation of the applies-to set. Lastly, we describe how the call resolution map is used to create the call graph.

```
1  <?php
2  function a() {
3    b();
4  }
5
6  function b() {
7    c();
8  }
9
10 function c() {
11   throw new Exception();
12 }
```

Listing 3.2: Exception flow example code

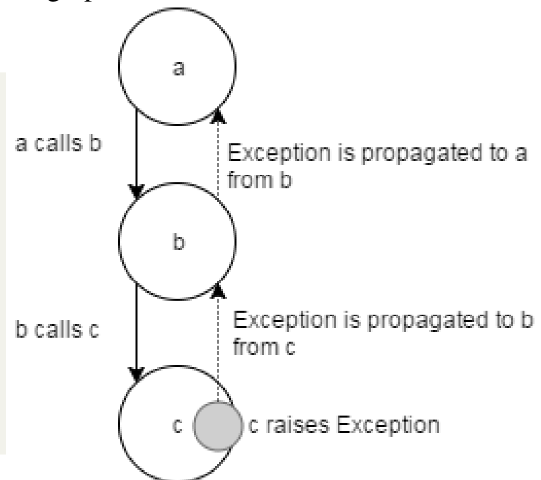



Figure 3.3: Visual representation of the exception flow of the program in Listing 3.2.

First, a partial order is created of all methods in the system. To create this partial order, we need an approach for comparing the methods we want to insert into it. Comparing two methods $a$ and $b$ can have four different outcomes:

1. $a$ and $b$ are equal, i.e. $a$ and $b$ are the same method.

2. $a$ and $b$ are not comparable, i.e. $a$ and $b$ will not have a relation in the partial order. This is the case when $a$ and $b$ do not have the same name, or are not defined in the same type hierarchy.

3. $a$ is greater than $b$, i.e. $a$ will be placed above $b$ in the partial order. Let $x$ and $y$ be the classes, interfaces or traits in which $a$ and $b$ respectively are defined. If $a$ and $b$ have the same name, and $x$ and $y$ are part of the same type hierarchy, and $x$ is a supertype of $y$, then $a$ is greater than $b$.

4. $a$ is smaller than $b$, i.e. $a$ will be placed below $b$ in the partial order. This is the exact opposite of the previous situation.

Let $m$ and $n$ be two methods that are to be compared. Let $x$ be the entity in which $m$ is defined, and let $y$ be the entity in which $n$ is defined. Algorithm 2 describes how two methods are compared in order to give them the correct place in the partial order. The algorithm operates on two methods $m$ and $n$. Two helper functions are presented as well: Algorithm 3 shows how a class can be compared to a trait and Algorithm 4 shows how two classes can be compared.

**Algorithm 2** Comparing two methods

```
Input: Two methods m and n
Output: One of the following: GREATER, SMALLER, EQUAL or NOT_COMPARABLE
 1: function COMPAREMETHODS(m, n)
 2:     x ← GETTYPEWHICHDEFINESMETHOD(m)
 3:     y ← GETTYPEWHICHDEFINESMETHOD(n)
 4:     if m.name ≠ n.name then
 5:         return NOT_COMPARABLE
 6:     else
 7:         if x.name = y.name then
 8:             return EQUAL
 9:         else if ISCLASSORINTERFACE(x) and ISCLASSORINTERFACE(y) then
10:             return COMPARECLASSES(x, y)
11:         else if ISCLASSORINTERFACE(x) and ISTRAIT(y) then
12:             return COMPARECLASSANDTRAIT(x, y)
13:         else if ISTRAIT(x) and ISCLASSORINTERFACE(y) then
14:             return INVERT(COMPARECLASSANDTRAIT(y, x))
15:         else if ISTRAIT(x) and ISTRAIT(y) then
16:             return NOT_COMPARABLE
17:         end if
18:     end if
19: end function
```

As can be seen from Algorithm 3, comparing two methods that are located in traits will always yield ‘not comparable’. When two traits, $t_1$ and $t_2$, both implement a certain method $m$ and are imported into the same class, this will result in an error as the PHP interpreter does not know which method $m$ it should prefer. This conflict can be resolved by using *trait adapters*, which are able to modify the name of an imported method. Implementing support for these adapters would require significant effort beyond our current scope, and therefore remains future work. Our personal experience is that these adapters are not used often.

After constructing the partial order, the call resolution map can be created, which is used when constructing the call graph. The call resolution map is a different representation of the applies-to sets which have been described in Section 2.4. The resolution map contains an entry for each possible call $t.m$. Here $t$ is a certain type and $m$ a certain method that can

**Algorithm 3** Comparing a class to a trait

**Input:** A class $c$ and a trait $t$
**Output:** One of the following: GREATER, SMALLER, EQUAL or NOT_COMPARABLE

```
 1: function compareClassAndTrait(c, t)
 2:     if c <: t then
 3:         return SMALLER
 4:     else if t <: c then
 5:         return GREATER
 6:     else
 7:         for all s ∈ classesUsingTrait(t) do
 8:             outcome ← compareClasses(s, c)
 9:             if outcome ≠ NOT_COMPARABLE then
10:                 return outcome
11:             end if
12:         end for
13:         return NOT_COMPARABLE
14:     end if
15: end function
```

**Algorithm 4** Comparing two classes

**Input:** Two classes (or interfaces) $c_1$ and $c_2$
**Output:** One of the following: GREATER, SMALLER, EQUAL or NOT_COMPARABLE

```
 1: function compareClasses(c1, c2)
 2:     if c1 <: c2 then
 3:         return SMALLER
 4:     else if c2 <: c1 then
 5:         return GREATER
 6:     else
 7:         return NOT_COMPARABLE
 8:     end if
 9: end function
```

be called on objects of that type, as well as methods that can be called on the type itself through static calls. The map contains for each entry a list of methods that the call $t.m$ can be resolved to. This call resolution map is created by doing a top-down breadth-first traversal of the partial order. When entering a node (i.e. a method) in the partial order, the following rules are applied.

A call $t.m$ can be resolved to:

1. Method $m$ in type $t$ if it is present and it is not abstract. Applying this rule is trivial.

2. Method $m$ in type $s$ if it holds that $s <: t$. This rule requires knowledge of the class hierarchy, and is implemented by adding the current method $m$ to all entries $s.m$ for which holds that $s <: t$.

3. Method $m$ in type $s$ if it holds that $t <: s$ and $t$ does not implement $m$ and there exists no type $u$ which implements $m$ for which holds that $u <: s$ and $t <: u$. In order to implement this rule, first all direct children $c.m$ of the current method in the partial order are fetched. Then, for each of these children, the class $s$ for which holds that $c <: s$ and $s <: t$ is fetched, and method $t.m$ is added to $s$ in the call resolution map.

When the call resolution map is created, the final call graph is created by traversing all scopes. For each method call $t.m$ in scope $s$ an edge is added from $s$ to all entries that are registered in the call resolution map for the call $t.m$. Note that the result is not a call graph in the classical sense of the word: although each edge ends in a method scope, an edge can start in a scope which is nested in a guarded scope, instead of starting in a method scope.

## 3.3 Flow Detection

Two types of flow can be discerned: the local flow and the inferred flow. The local flow comprises all exceptions that are not entering the current scope via another scope. It is thus the combination of the *raises* and the *generates* set (see Section 2.2.1). The *raises* set is computed by traversing the AST and gathering all `throw` statements in a scope and adding the types of the expressions that follow those statements to the *raises* set of that scope.

The second type of flow is the inferred flow. This flow contains all exceptions that enter a scope via another scope. It can be constructed by computing the union of the *uncaught* set and the *propagates* set. If a scope does not contain a nested guarded scope, the uncaught set is always empty. If it does contain a guarded scope $g$, the exceptions that are encountered in $g$ need to be determined first. If those are known, then for each exception $e$ that is encountered in $g$ it has to be determined whether $e$ is caught or not. Let $type(e)$ denote the type of $e$, let $catches(c)$ denote the type that is caught by catch clause $c$ and let $C$ be the set of all catch clauses of guarded scope $g$. Algorithm 5 describes how to compute the set of exceptions that are encountered in but not caught by a guarded scope $g$.

Algorithm 6 contains a description of the computation of the *propagates* set. Here, the call graph that was described in Section 3.2 is used: the `resolveCallToScopes` function fetches all scopes that can be called by instruction $i$.

**Algorithm 5** Computing the *uncaught* set

**Input:** A guarded scope $g = (n, C)$
**Output:** The *uncaught* set of this guarded scope

```
 1: function DETERMINEUNCAUGHT(g = (n, C))
 2:     uncaught ← ∅
 3:     for all e ∈ encounters(n) do
 4:         for all c ∈ C do
 5:             if type(e) <: catches(c) then
 6:                 continue
 7:             end if
 8:         end for
 9:         uncaught ← uncaught ∪ {e}
10:     end for
11:     return uncaught
12: end function
```

**Algorithm 6** Computing the *propagates* set

**Input:** A guarded scope $s = (I, G)$
**Output:** The *uncaught* set of this guarded scope

```
 1: function DETERMINEPROPAGATES(s = (I, G))
 2:     propagates ← ∅
 3:     for all i ∈ I do
 4:         if ISCALLINSTRUCTION(i) then
 5:             for all m ∈ RESOLVECALLTOSCOPES(i) do
 6:                 propagates ← propagates ∪ encounters(m)
 7:             end for
 8:         end if
 9:     end for
10:     return propagates
11: end function
```

The *propagates* and *uncaught* set algorithms assume that in the scopes that are referred to (called scopes and nested guarded scopes respectively) all exceptions that can be encountered are known. At the start of the computation, this is not the case. To work around this problem, we devised a fixed-point algorithm, which can be found in Algorithm 7. First, for all scopes in the system, the local flow is computed. All these scopes are also added to the worklist. As long as the worklist is not empty, a scope is fetched and the inferred flow is computed for that scope. Thereafter, we verify whether the flow for that scope has changed. If so, we add all the scopes that are influenced directly by $s$ to the worklist. A scope $u$ is affected by another scope $v$ when it holds that either $v$ is enclosed by $u$ or $u$ calls $v$. The algorithm terminates when there are no scopes left on the worklist. For each scope in the analysed system, the exceptions that can be encountered in that scope are contained in the *encounters* set.

**Algorithm 7** Fixed-point algorithm for determining the exception flow

**Input:** A set of scopes $S$
**Output:** None, encounters set is added as an attribute on each scope

```
1: function DETERMINEFLOW(S)
2:     worklist ← ∅
3:     for all s ∈ S do
4:         DETERMINELOCALFLOW(s)
5:         ENQUEUE(worklist, s)
6:         for all nested_scope ∈ GETALLNESTEDSCOPES(s) do
7:             DETERMINELOCALFLOW(nested_scope)
8:             ENQUEUE(worklist, nested_scope)
9:         end for
10:    end for
11:    while worklist ≠ ∅ do
12:        s ← DEQUEUE(worklist)
13:        DETERMINEINFERREDFLOW(s)
14:        if SCOPEHASCHANGED(s) then
15:            for all affected_scope ∈ GETAFFECTEDSCOPES(s) do
16:                ENQUEUE(worklist, affected_scope)
17:            end for
18:        end if
19:    end while
20: end function
```

A pure PHP implementation of the approach presented in this section can be found at `http://github.com/tomdenbraber/php-exception-flow`.

## 3.4 Research Scope

Although the *encounters* set presented in Section 2.2 consists of four subsets, we model only three of them: *raises*, *propagates* and *uncaught*, but not *generates*. As we focus on the exception flow induced by the programmer, we have left the *generates* set out of scope. This decision is supported by the fact that one of the shortcomings of the approach by Robillard and Murphy [26] was that the *generates* contained information that was too imprecise to their liking. They argue that semantic analysis is needed to improve the precision of the *generates* set.

We also limit the *raises* set: exceptions that are thrown by PHP library functions are ignored. The only source of information for the *raises* set of PHP library functions is the documentation, which is excluded from our research scope. As the PHP manual states that most functions use the internal error system of PHP instead of exceptions, this decision has limited impact on the accuracy of our approach [3].

The third restriction of our scope is related to a specific feature of PHP, namely *error handlers*. PHP functions can raise errors or warnings, which differ from normal exceptions.

Programmers can write custom error handlers which are called when such an error occurs. These error handlers are normal functions, and can be used for e.g. handling the error, or transforming the error into an exception. To be able to detect this kind of exceptions, the error handlers should be linked to the functions that can generate errors. This would involve parsing the documentation.

Another aspect which does influence the exception flow but which has not been included in this research is the resolution of 'dynamic' method and function calls. In PHP, functions can be called via variables, or via other functions. Two examples can be found in Listing 3.4. To be able to resolve these kind of calls, it would be necessary to track the actual values of variables. Because that is a completely different topic, we decided to remove dynamic call resolution from the scope of this research.

```
1  <?php
2  //EXAMPLE 1
3  $fn = "a";
4  $$fn(); //leads to a function call to a
5  //EXAMPLE 2
6  call_user_func("a"); //leads to a function call to a
```

Listing 3.4: Dynamic calls

The last limitation of our approach is related to the PHP-CFG project, as it has only a rudimentary interpretation of exception handling constructs. A `throw` statement is interpreted as an empty `return` statement, and all code located in `catch` and `finally` blocks is ignored. This choice was made as the developers on the project could not reach an agreement about the specific implementation of intraprocedural exception flow [7].

[7] See `https://github.com/ircmaxell/php-cfg/pull/23` for the discussion on this matter.

# Chapter 4

# Algorithm Accuracy

Because of the limitation of our scope (see Section 3.4), we know that our approach will both be unsound and incomplete. In this chapter, we evaluate how well the approach as described in Chapter 3 works in practice, to see what the implications are of the limitations of our scope.

We perform the evaluation by means of two research questions:

**RQ1. How sound is the approach?**
With the answer to the this question, we can know to what extent the output of the approach can be trusted. If the output is sound, we know that for every exception that the algorithm detects in a method, that this exception is also encountered in the method in the actual situation.

**RQ2. How complete is the approach?**
Knowing the answer to this question enables us to say whether the exceptions that the algorithm detects in a method are all the exceptions that the method encounters in practice, or whether there might be more exceptions which were missed by the algorithm.

The approach relies on several existing tools. We know that these tools are unsound, and the effects of the unsoundness are made explicit by this evaluation. We evaluate our approach by manually building three oracles. Section 4.1 introduces the process of building and using an oracle in Section, and contains a brief description of the projects for which an oracle was created. Thereafter, we present the results of the evaluation in Section 4.2. Our conclusion concerning the evaluation of the approach can be found in Section 4.3.

## 4.1 Methodology

### 4.1.1 Oracle usage

An oracle describes the ‘truth’: it contains valid information about the real world. The approach as presented in Chapter 3 is a model, and needs validation. The difference between

the oracle and the output of the approach gives us information about the soundness and the completeness of our model.

First, all the mismatches between the output of the algorithm and the oracle are manually analysed. The mismatches are categorized by their origin (i.e. *raises*, *propagates*, or *uncaught*). Each error is marked as either a false negative or a false positive. For each error it is then checked why this error occurred, and if it is a *real* error or a *propagated* error. An error is marked as real if the reason why the exception was not detected correctly can be found in the scope in which the error occurred. In all other cases, the error is marked as propagated. As an example, consider the code in Listing 4.1. For this program, the oracle can be found in Table 4.1. Assume that the *raises* set for the `throwSomething` method is computed correctly by the algorithm, but that the *propagates* set for the `with_real_error` function is empty, in other words: the algorithm failed to detect that the expression $\$a \rightarrow throwSomething$ propagates a `RuntimeException` into `with_real_error`. The *propagates* set for the `with_propagated_error` function will now always contain an error, even if the call to `with_real_error` is resolved correctly, as the oracle says that a `RuntimeException` should be found. When examining the errors of the algorithm, the error concerning the `with_real_error` function is marked as a *real* error. The error in `with_propagated_error` is marked as a *propagated* error if the call to `with_real_error` was resolved correctly, otherwise it is categorized as a *real* error.

```php
<?php
class A {
  public function throwSomething() {
    throw new RuntimeException();
  }
}

function with_real_error(A $a) {
  $a->throwSomething();
}

function with_propagated_error() {
  $a = new A();
  with_real_error($a);
}
```

Listing 4.1: Real errors and propagated errors

### 4.1.2 Oracle creation

An oracle consists of a random selection of methods and functions (scopes) from a system. Its structure closely resembles the output format of the algorithm, which is a list of scopes and for each scope a list of exceptions grouped by their origin, namely the different sets mentioned in Section 2.2.1.

| Scope | Exception set | Propagated from | Exception type |
|---|---|---|---|
| `A::throwSomething` | *raises* | n/a | `RuntimeException` |
| `with_real_error` | *propagates* | `A::throwSomething` | `RuntimeException` |
| `with_propagated_error` | *propagates* | `with_real_error` | `RuntimeException` |

Table 4.1: An example oracle corresponding to Listing 4.1

For each scope $s$ in the oracle, the *throws*, *propagates* and *uncaught* sets are manually defined. The *propagates* set for $s$ depends on all the function and method calls that $s$ contains. Let $S$ be the collection of all scopes in a system. For each randomly selected method scope $s$, all scopes $\{t \in S | s \text{ calls } t\}$ are also included in the oracle. An oracle thus consists of a set of randomly selected scopes, but to be able to model the *propagates* set, scopes on which a randomly selected scope has a call dependence are also included.

The scopes in the call dependence set of a scope cannot always be determined, even when manually constructing the oracle, as some function calls are dynamic (see Section 3.4). Often, the functions and methods that are called via these dynamic constructs can be deduced from the immediate context (i.e. the scope that contained the call or the class that contained the scope). In these cases, we decided to include the called scopes in the oracle. In all other cases, the dynamic call was not resolved and was left out of the oracle. Even though dynamic calls are left out of the approach, they are included in the oracle, which enables us to see their impact.

If a scope in the call dependence set of a certain scope is defined in the PHP library, its definition was looked up on the PHP website[1]. If the documentation states that that function could raise an exception, this exception was added to the *propagates* set of the caller method. Here, the same holds as for dynamic calls: although it is evident that the algorithm will fail in all these cases, it is interesting to see how often PHP library functions might cause an exception to enter the system.

Another important decision that was made also relates to the *propagates* set. Consider a scope $s$ which calls scopes $t$ and $u$, and both $t$ and $u$ encounter a certain exception of type $x$. An exception of type $x$ has thus two paths to scope $s$: the path via the call to $t$ and a path via the call to $u$. This situation is visualised in Figure 4.2. The output of the algorithm could be that $s$ encounters an exception $e$ of type $x$, because it is contained in $propagates(s)$. However, if this is all the information, it cannot be decided whether the algorithm propagated $e$ from $t$, from $u$, or from both $t$ and $u$. Even if the outcome of the algorithm would say that $s$ propagates an exception of type $x$, it might still be incorrect. We therefore include all possible edges in the propagation set of the oracle.

For the situation in Figure 4.2, $propagates(s)$ would contain both $t \rightarrow x$ and $s \rightarrow x$. Here, $\rightarrow$ denotes that the scope on the left-hand side propagates an exception of the type written on the right-hand side of the $\rightarrow$ symbol.

[1] See `http:///www.php.net`

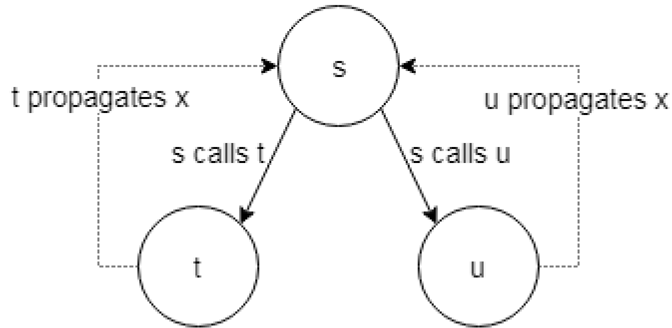



Figure 4.2: Example situation for exception entering scope via different routes

| Project | LOC[3] | #classes[4] | #methods[5] | #throw stmts | #try stmts | #catch stmts |
|---|---|---|---|---|---|---|
| Fabric | **62914** | 1024 | 3904 | **938** | 94 | 126 |
| Flarum | 12956 | 353 | 1175 | 61 | 14 | 13 |
| Monolog | 5956 | 100 | 521 | 87 | 4 | 4 |

Table 4.2: Characteristics of the projects that are used to create oracles

### 4.1.3 Corpus

The corpus for evaluating the approach has to consist of programs with different use cases, applications and origins. By testing the approach on such a corpus, we show that the approach can be applied on a wide range of projects while yielding comparable results. Based on this criterion, three projects were selected. To start with, we selected Fabric, a framework produced by our industry partner. Subsequently, we chose Flarum, an application written on the Laravel[2] framework. We decided that the last project to be used was Monolog, a stand-alone package that is used for sending logs over a whole range of mediums.

A list of characteristics relevant for our research can be found in Table 4.2. These characteristics have been collected by counting the nodes in the AST unless stated otherwise.

#### Fabric

Fabric is the framework that is produced by our industry partner and is used as a basis for all their web applications. It contains a lot of functionalities: logging, session management, and database interaction among others. It is a closed-source project and has been under development for 7 years. Over those years, 22 different programmers have worked on the

[2]Laravel is a framework for developing PHP applications. See `https://www.laravel.com`.
[3]As counted by cloc, see `https://github.com/AlDanial/cloc`
[4]Also includes interfaces and traits
[5]Also includes abstract methods

system. As can be seen from Table 4.2, it is the largest of the three projects. In comparison with the other projects, it contains a relatively high number of throw statements.

#### Flarum

The second project for which we created an oracle is called Flarum[6]. Flarum is an application which can be used to build forums and has been built on Laravel. At the time of writing, Laravel is a popular framework used for developing PHP web applications [1, 5]. The project was chosen because it is of considerable size, has close to 50 contributors and was written on a popular framework. The framework itself was not incorporated in the oracle: all calls to methods or functions that reside in the framework have been left out of the oracle. However, the framework does influence the architecture of the application, as the application needs to be plugged into the framework to make use of its features. It is important to note that the philosophy of Laravel differs a lot from the philosophy of Fabric. For example, applications which are built on Fabric mostly use dependency injection to make use of the framework, whereas Laravel uses "static facades" to make dependencies available. These facades are available from everywhere in the system.

#### Monolog

The last project for which we built an oracle is called Monolog[7]. Monolog is a package that provides functionalities for logging. It offers handlers for a wide range of mediums to which logged messages can be sent. Monolog is neither a framework nor a web application, but a stand-alone package that can be plugged into existing projects to take care of sending logs. According to Packagist[8], almost 2500 other projects depend on Monolog [2]. More than 250 developers have contributed to Monolog over the past 6 years. Monolog is dissimilar to both other projects. It differs from Fabric in the sense that it only offers one functionality and it does not enforce or encourage a certain type of architecture. It differs from Flarum because there is no system or dependency which influences its architecture: it does not have to adhere to certain interfaces or subclass certain framework classes.

## 4.2 Results

After creating an oracle for each of the applications described in Section 4.1.3 and categorising all the errors, we can see how well the approach works in practice. First, the precision and the recall for each of the created oracles are presented. Thereafter, for each of the oracles, the results will be explained briefly.

### 4.2.1 Precision and recall

Table 4.3 presents the results of the evaluation. The numbers that are noted behind the

[6] Source can be found at `https://github.com/flarum/core`
[7] Source can be found at `https://github.com/Seldaek/monolog`
[8] Packagist is a package repository for PHP packages. See `https://packagist.org`.

| | Fabric | Flarum | Monolog | Aggregated |
|---|---|---|---|---|
| Oracle statistics | | | | |
| #methods | 251 | 360 | 243 | 854 |
| #call edges | 980 | 597[9] | 669[9] | 2246 |
| *encounters* | | | | |
| algorithm | 1370 | 71 | 298 | 1739 |
| oracle | **2087** | 89 | 316 | 2492 |
| intersection | 1247 | 71 | 262 | 1580 |
| precision | 0.91 | **1.0** | 0.88 | 0.91 |
| recall | 0.60 | 0.80 | **0.83** | 0.63 |
| *raises* | | | | |
| algorithm | 53 | 18 | 28 | 99 |
| oracle | 53 | 18 | 28 | 99 |
| intersection | 53 | 18 | 28 | 99 |
| precision | **1.0** | **1.0** | **1.0** | **1.0** |
| recall | **1.0** | **1.0** | **1.0** | **1.0** |
| *propagates* | | | | |
| algorithm | 1310 | 53 | 269 | 1632 |
| oracle | 2017 | 71 | 287 | 2375 |
| intersection | 1191 | 53 | 234 | 1478 |
| precision | 0.91 | **1.0** | 0.87 | 0.91 |
| recall | 0.59 | 0.75 | **0.82** | 0.62 |
| *uncaught* | | | | |
| algorithm | 7 | 0 | 1 | 8 |
| oracle | 17 | 0 | 1 | 18 |
| intersection | 3 | 0 | 0 | 3 |
| precision | 0.43 | n/a | 0.0 | 0.38 |
| recall | 0.17 | n/a | 0.0 | 0.17 |

Table 4.3: Evaluation results

cells with 'algorithm' and 'oracle' denote the number of exceptions contained in the output of the algorithm and the oracle respectively. The 'intersection' rows denote the number of exceptions that are both in the output of the algorithm and in the oracle, i.e. the intersection contains the true positives. The precision is then calculated as $\frac{\text{intersection}}{\text{algorithm}}$ and the recall as $\frac{\text{intersection}}{\text{oracle}}$.

### 4.2.2 Reasons for errors

To be able to better understand the results, each error (either a false positive or a false negative) was labeled with a reason. This reason describes why this error occurred. We

[9]Calls to methods in the underlying framework or external dependencies have been left out

found the following reasons:

I. A method could not be resolved because the type inference tool could not detect a precise type for the callee object;

II. A dynamic call to a method or function could not be resolved;

III. The exception to be detected is thrown by a PHP library function;

IV. The `throw` statement is located in a catch statement;

V. The exception to be detected has been caught and rethrown afterwards;

VI. A method is erroneously resolved.

We completely separate reason IV from reason V: reason IV occurs because of the fact that no types are inferred in catch clauses, whereas reason V occurs because PHP-CFG does not properly support `try`/`catch` constructs (see Section 3.4). The errors are not evenly distributed over the reasons, as some reasons occur more than others. Table 4.4 presents the number of errors for each reason.

| Reason | *encounters* | | *propagates* | | *uncaught* | |
|---|---|---|---|---|---|---|
| | Number | % | Number | % | Number | % |
| I. | 537 | **50.1** | 531 | **50.5** | 6 | 30.0 |
| II. | 72 | 6.7 | 72 | 6.9 | 0 | 0.0 |
| III. | 52 | 4.9 | 52 | 4.9 | 0 | 0.0 |
| IV. | 133 | 12.4 | 121 | 11.5 | 12 | **60.0** |
| V. | 2 | 0.2 | 0 | 0.0 | 2 | 10.0 |
| VI. | 19 | 1.8 | 19 | 1.8 | 0 | 0.0 |
| I., II. | 157 | 14.7 | 157 | 14.9 | 0 | 0.0 |
| I., III. | 21 | 2.0 | 21 | 2.0 | 0 | 0.0 |
| I., II., III. | 78 | 7.3 | 78 | 7.4 | 0 | 0.0 |

Table 4.4: The number of errors per reason

### 4.2.3 Fabric

The Fabric oracle was the largest of the three oracles in terms of number of encountered exceptions. We can see a difference between the *raises* set and the other two sets: the *raises* set is computed correcly, whereas both the *propagates* set and the *uncaught* set are not. For both *uncaught* and *propagates* it holds that the precision is higher than the recall.

Table 4.6 shows how often each of the reasons listed in Section 4.2.2 occurs for each exception set. If a reason does not occur at all, it is left out of the table. A combination of reasons can occur if there are multiple ways in which an exception of a certain type

can reach a function. One of the first things to note is that incomplete or too generic type information is the most common cause for a false negative in the Fabric oracle. A lot of errors were also due to propagation: the error did not occur in the enclosing scope, but was propagated via a call. Because missing type information was such a dominant factor, we separately analysed each individual non-propagated error due to type inference. In Table 4.5, we show which causes lead to incomplete type information.

| Cause | Total | % |
|---|---|---|
| Missing type annotation | 242 | **91.3** |
| Incorrect type annotation | 10 | 3.7 |
| Unresolved static property | 6 | 2.3 |
| Unresolved return type | 7 | 2.7 |

Table 4.5: Causes of incomplete type information

We can see that there is one dominant cause: in 91.3% of the errors due to type inference, a type could not been inferred on a property because of a missing type annotation[10]. Because the inference algorithm is dependent on these annotations for resolving the types of properties (see Section 3.1), methods called on these non-annotated properties could not be resolved. This leads to an error if such an unresolved method propagates an exception.

The precision and the recall for the *uncaught* set are both quite low. This can be explained by the fact that the PHP-CFG library does not analyse the code within a `catch` block. This problem is not inherent to our approach, as it can be resolved in the future without adapting the other algorithms of our approach. However, implementing a correct representation of `catch` blocks in the PHP-CFG library is not trivial. As a starting point, the work of Amighi et al. [7] could be used to implement a correct CFG creation algorithm for programs with exceptions.

If an exception is thrown from a `catch` block, the type can not be deduced correctly, as the type inference is based on the CFG. This knowledge can be used to explain both reason IV and reason V and all the false positives for the *uncaught* set, as the algorithm then puts an exception of type ‘unknown’ in the *uncaught* set. Because this ‘unknown’ exception can consequently be propagated, the *propagates* set also contains a lot of ‘unknown’ exceptions. In 97.4% of the false positives in the *propagates* set, the original cause can be deduced to an exception which is (re)thrown from a catch clause. The other false positives were due to erroneously resolved method calls (reason VI).

### 4.2.4 Flarum

The *throws* set was computed correctly. The *uncaught* set was empty in both the oracle and the output of the approach, due to the fact that there were no `try`/`catch` constructs in

[10] A type annotation is a comment in a specific format, which provides extra information to the programmer and can also be used for static analysis, such as type inference.

[11] The percentage of exceptions for the reason stated in the left-most column that has been propaged

| Reason | *propagates* | | | | *uncaught* | | | |
|---|---|---|---|---|---|---|---|---|
| | Total | | Error propagated | | Total | | Error propagated | |
| | Number | % | Number | %[11] | Number | % | Number | %[11] |
| I | 484 | **58.6** | 265 | 54.8 | 6 | 42.9 | 3 | 50.0 |
| II | 70 | 8.5 | 38 | 54.3 | 0 | 0.0 | n/a | n/a |
| III | 16 | 1.9 | 10 | 62.5 | 0 | 0.0 | n/a | n/a |
| IV | 0 | 0.0 | n/a | n/a | 6 | 42.9 | 0 | 0.0 |
| V | 0 | 0.0 | n/a | n/a | 2 | 14.2 | 0 | 0.0 |
| I, II | 157 | **19.0** | 157 | **100.0** | 0 | 0.0 | n/a | n/a |
| I, III | 21 | 2.5 | 21 | **100.0** | 0 | 0.0 | n/a | n/a |
| I, II, III | 78 | 9.4 | 78 | **100.0** | 0 | 0.0 | n/a | n/a |

Table 4.6: Reasons for false negatives in the Fabric oracle

the analysed methods. With a precision of 1.0 and a recall of 0.80, there were only false negatives in the output of the approach. All false negatives originated from the *propagates* set because of two reasons, as can be seen in Table 4.7. A large portion of the errors is again due to incomplete type information.

| Reason | *propagates* | | | |
|---|---|---|---|---|
| | Total | | Error propagated | |
| I | 12 | **66.7** | 1 | 8.3 |
| III | 6 | 33.3 | 5 | **83.3** |

Table 4.7: Reasons for false negatives in the Flarum oracle

### 4.2.5 Monolog

The recall of the approach on the Monolog project was better than the recall on the other projects, mostly due to a better performance on the propagates set. This is the case because the project contains a lot of type hints, which enabled the type inference tool to reconstruct a relatively high amount of types. The errors were mostly due to exceptions that were thrown by PHP functions, as can be seen in Table 4.8. We can see that the number of propagated errors is relatively high, as for the other projects.

In comparison with the other oracles, the precision of the Monolog oracle is somewhat lower: 3% lower than Fabric's precision and 12% lower than the precision of Flarum. Upon further investigation, the false positives could be divided over the same set of reasons as presented in Section 4.2.2. This classification can be found in Table 4.9. In the other oracles, reason VI was almost absent, but for Monolog, it is one of the most occurring reasons for

| Reason | *propagates* | | | | *uncaught* | | | |
|---|---|---|---|---|---|---|---|---|
| | Total | | Error propagated | | Total | | Error propagated | |
| | Number | % | Number | %[11] | Number | % | Number | %[11] |
| I | 19 | **35.8** | 5 | 26.3 | 0 | 0.0 | n/a | n/a |
| II | 2 | 3.8 | 1 | 50.0 | 0 | 0.0 | n/a | n/a |
| III | 30 | **56.6** | 28 | 93.3 | 0 | 0.0 | n/a | n/a |
| IV | 2 | 3.8 | 2 | 100.0 | 1 | 100.0 | 0 | 0.0 |

Table 4.8: Reasons for false negatives in the Monolog oracle

false positives. The reason for an erroneously resolved method is that the type inference algorithm does not make a distinction between types inferred from parameter declarations and types that can be inferred from a constructor call. Consider the code in Listing 4.3. The type inference tool would now deduce that variable `$a` has type `A` and that variable `$b` has type `B`. While these types are both correct, it does not make the distinction that `$a` could be of type `A`, but also of all types that are a subtype of `A`, whereas `$b` can be of type `B` and `B` only. In some cases, a method call is thus incorrectly resolved to a method defined in a subtype.

```
1 <?php
2 function(A $a) {
3   $b = new B();
4 }
```

Listing 4.3: Type inference limitations

| Reason | *propagates* | | | | *uncaught* | | | |
|---|---|---|---|---|---|---|---|---|
| | Total | | Error propagated | | Total | | Error propagated | |
| | Number | % | Number | %[11] | Number | % | Number | %[11] |
| I | 16 | **45.7** | 5 | 31.3 | 0 | 0.0 | n/a | n/a |
| IV | 3 | 8.6 | 3 | 100.0 | 1 | 100.0 | 0 | 0.0 |
| VI | 16 | **45.7** | 0 | 0.0 | 0 | 0.0 | n/a | n/a |

Table 4.9: Reasons for false positives in the Monolog oracle

## 4.3 Concluding remarks

The aggregated recall of the *encounters* set of the approach is 0.63, whereas the aggregated precision of the *encounters* set is 0.91. Because the precision and recall are both smaller

than 1.0, it is possible to argue that our approach is both unsound and incomplete. However, the false negatives in the approach all come from external sources, i.e. the exception flow algorithm itself does not add exceptions where they should not be. Due to incorrect type annotations, incomplete type information and an incomplete CFG implementation, exceptions with an "unknown" type are sometimes detected in methods where they might not be encountered in the actual code. This is not due to the exception flow algorithm itself. We conclude that the exception flow algorithm is sound, but that the underlying dependencies are not. The incompleteness of the approach is partly due to libraries which generate incomplete output (i.e. the type inference algorithm), and partly because of the limitation of the research scope (e.g. dynamic calls, library functions).

When looking at the results for the different oracles, a few aspects need to be discerned. The first is that 'type inference' takes quite a large portion of the errors, as reason I, IV, V and (to a lesser extent) reason VI have all to do with type inference. If we accumulate the number of errors for these reasons, we can see that these account for 64.5% of the total number of errors. This gives a strong indication that an improvement in the type inference algorithm directly leads to a better performance of the exception flow detection algorithm. A possible improvement would be to use another source of information for resolving the types of properties. Types of properties are now resolved by their annotations, and a property without annotation always leads to incomplete type information. The type inference algorithm could incorporate write operations to properties to detect the types of not-annotated properties.

A second observation is that reason II and III, which were both decided to be out of scope, account for a smaller yet significant portion of the errors. Adding support so that errors caused by reason II and III will be prevented, will improve the recall of the algorithm. Lastly, it is also clear that the *propagates* set is the largest in all three oracles: most exceptions enter a scope not because they are uncaught by a nested scope or raised directly in a scope, but via a call to another scope. Conversely, the number of exceptions in the *uncaught* set is very small in all three oracles. The approach does not perform well on this set, with the highest precision and recall being only 0.43 and 0.17 respectively. However, these low numbers do not strongly influence the overall performance due to the small size of the *uncaught* sets.

# Chapter 5

# Empirical Study

Using the presented algorithm, we can analyse a number of software projects to discover the exception handling practises that are used in the PHP community. To start with, Section 5.1 presents a set of research questions. Subsequently, we introduce the corpus of projects which are used for the analysis. Section 5.3 explains how each of the research questions will be answered. The chapter closes with a description of the threats to the validity of this study.

## 5.1 Research Questions

**RQ1. How many exceptions does a method typically encounter?**
For PHP systems, we currently do not have any information concerning the presence of exceptions in methods. If a method encounters many exceptions, this might signal a problem: the developer who wrote the method is perhaps not aware of all these exceptions.

**RQ2. How close to the source are exceptions caught?**
It is considered a good practice to catch exceptions as close to its original scope as possible, so that there is enough information available to handle the exception [30]. Catching an exception that has been propagated through many scopes might indicate that the catch clause catches an exception which it was not intended to catch.

**RQ3. What is the typical distance in the class hierarchy between a caught exception and the type which is used in the catch clause type that catches it?**
An exception can be caught by its exact type, but also by any of its supertypes. Barbosa et al. [8] found that the use of an overly-generic catch clause lead to bugs in multiple projects. It is therefore a good practice to catch an exception by a type which is as close to the type of the thrown exception as possible. Using a more precise type also gives the opportunity to handle the direct cause of the exception.

**RQ4. How often are exceptions documented correctly?**
In PHP, all exceptions are unchecked: exceptions are not part of a method's signature.

However, it is possible to use phpdoc to document exceptions that are encountered in a method or function. An example of such an annotation can be found in Listing 5.1. A documented exception is more visible than a non-documented exception, and is therefore likely to be handled correctly more often than a non-documented exception. If a method encounters an exception of type *t* and has a `@throws` annotation for exactly type *t*, we consider the exception as documented correctly. We are also interested in the usage of "contractual annotations". If an abstract super method does not document an exception, we expect that its implementations do not propagate any exceptions, because in that case, the implementation would break the contract.

```
1 <?php
2 class A {
3   /**
4     * @throws SomeException
5     */
6   public function someFunction() {
7     throw new SomeException();
8   }
9 }
```

Listing 5.1: A method with a `@throws` annotation

## 5.2 Corpus

Our corpus consists of a selection of Open Source Software (OSS) projects in combination with three projects that were created by our industry partner. We selected the OSS projects using OpenHub[1], which monitors a large number of OSS projects. Our corpus includes a subset of the 50 most popular projects which have PHP as their main language. Because our computational resources were limited, projects which contained more than 500000 lines of PHP code could not be analysed. By inspecting the type, size, use cases and origin of projects we have created a diverse corpus, consisting of both open and closed source projects.

Before we analysed the projects, we collected some general information for each project. This information can be found in Table 5.1. For each project, we installed all required dependencies (except those dependencies that were needed for development). We manually analysed the projects to find a namespace or prefix which separates the project from its dependencies. The LOC2 column represents the number of lines of PHP code of the project, including its dependencies. All the other data that Table 5.1 presents is taken from the project itself, excluding its dependencies. Both LOC metrics only include PHP lines of code as counted by cloc[2]. For some projects, it was not possible to find a namespace that

[1]See `https://www.openhub.net/tags?names=php`
[2]See `https://github.com/AlDanial/cloc`.

| Project | Version | LOC1 | LOC2 | #class | #interface | #method | #function | #throw | #try | #catch | #`@throws` |
|---|---|---|---|---|---|---|---|---|---|---|---|
| CakePHP | 3.4.9 | 56111 | 61549 | 399 | 27 | 4374 | 0 | 402 | 62 | 64 | 388 |
| CodeIgniter | 3.1.5 | 28879 | 29319 | 133 | 0 | 1510 | 0 | 13 | 8 | 8 | 0 |
| *Digitaalloket* | 7.3 | 16032 | 78296 | 297 | 70 | 1239 | 0 | 61 | 35 | 51 | 17 |
| Doctrine | 2.5.6 | 29587 | 105930 | 342 | 25 | 2431 | 0 | 329 | 13 | 13 | 179 |
| DokuWiki* | 2017-02-19b | n/a | 171067 | 320 | 3 | 3210 | 587 | 202 | 29 | 30 | 93 |
| Drupal | 8.3.4 | 82527 | 461714 | 2096 | 465 | 13327 | 74 | 820 | 250 | 266 | 409 |
| *Fabric* | 1.6 | 67205 | 67205 | 781 | 247 | 3922 | 19 | 938 | 94 | 126 | 321 |
| Joomla | 3.7.2 | 208197 | 208197 | 1340 | 30 | 7667 | 61 | 824 | 537 | 550 | 867 |
| *Objectbrowser* | 2.2 | 5437 | 58849 | 71 | 2 | 311 | 0 | 10 | 10 | 10 | 17 |
| Nette | 2.4.6 | 12100 | 12100 | 120 | 21 | 952 | 0 | 212 | 27 | 34 | 93 |
| PEAR | 1.10.4 | 30986 | 30986 | 73 | 1 | 1119 | 0 | 3 | 0 | 0 | 3 |
| Phing* | 2.16.0 | n/a | 49257 | 497 | 19 | 4452 | 2 | 1135 | 199 | 211 | 671 |
| phpBB | 3.2.0 | 66475 | 445198 | 684 | 42 | 4070 | 127 | 271 | 79 | 91 | 160 |
| phpDocumentor2 | 2.9.0 | 19184 | 196184 | 317 | 31 | 1451 | 0 | 92 | 15 | 16 | 67 |
| PHPUnit | 6.2.3 | 17110 | 61030 | 145 | 22 | 1035 | 0 | 257 | 49 | 70 | 133 |
| Roundcube* | 1.3.0 | n/a | 133628 | 245 | 4 | 2605 | 198 | 138 | 47 | 48 | 141 |
| Smarty | 3.1.30 | 17092 | 18194 | 172 | 0 | 898 | 57 | 105 | 11 | 11 | 127 |
| Symfony | 3.3.3 | 256848 | 287204 | 1618 | 276 | 9093 | 2 | 2017 | 361 | 360 | 914 |
| Wordpress | 4.8 | 178583 | 178583 | 366 | 6 | 4013 | 3075 | 141 | 52 | 53 | 39 |
| Zend Framework | 2.4.12 | 158688 | 164151 | 1814 | 341 | 12258 | 0 | 2730 | 176 | 183 | 2249 |

Table 5.1: Characteristics of the projects which will be used in the empirical study. Projects in *italics* are closed source projects. Projects marked with an * are analysed including the dependencies.

separated the project from its dependencies. In those cases, we added an * as a suffix to the project name, and the presented data includes the dependencies. The projects that are printed in *italics* are the projects which originate from our industry partner.

## 5.3 Methodology

Before gathering the data needed for answering the questions as presented in Section 5.1, we first build the prerequisite resources as listed below.

**exception flow** The exception flow is calculated using the algorithm presented in Chapter 3. This resource contains for each scope in the system all the exception that it encounters, as well as the causes for those exceptions (i.e. if they are raised, propagated or uncaught).

**`@throws` annotations** For answering the question concerning the documentation of exceptions, we build a set which contains the `@throws` annotations for each method or function in the system under analysis. This is done by traversing the AST of the system, collecting all the methods and parsing the documentation using phpDocu-

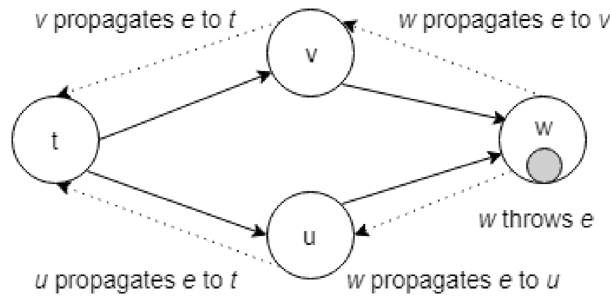



Figure 5.2: An example where the exception can be propagated from two sources

mentor[3].

**method order** The partial order of all the methods in the system (see Section 3.2). It contains for each method in the system the methods that override it.

**paths to catch clauses** This set contains for each caught exception *e* the shortest path from the initial scope of *e* to the guarded scope in which *e* is caught. The set is computed by first registering for each catch clause *c* all the exception that are caught by *c*. Then, for each of these exceptions, the shortest path between the original scope and the guarded scope which contains the catch clause is calculated using Dijkstra's algorithm [14].

**type hierarchy** For each type *t* in the system, this resource contains the types to which *t* can be resolved, and which types *t* resolves itself.

We will now detail the methodology for answering each of the questions as posed in Section 5.1.

**RQ1.** For each method, we record the number of exceptions that it encounters. Thereafter, we group methods with the same number of encountered exceptions. If an exception can enter a method via different paths, it is counted only once. This situation is visualised in Figure 5.2. Exception *e* can enter scope *t* via both scope *u* and scope *v*. Because the exceptions that are encountered by *u* and *v* both refer to the same exception *e* which is coming from scope *w*, the number of encountered exceptions is 1 for all scopes in this example.

**RQ2.** This analysis is based on the path to catch clauses set. As this set contains all the shortest path from the catch clause in which an exception is caught to the scope in which it was raised, we can find the path lengths by counting all the scopes in the path between the initial scope and the catch clause. Figure 5.3 visualises two paths from the same exception to the same guarded scope. Here, the path via scope *b* is the shortest and will be registered with length 3. We then aggregate the number of paths by their lengths.

[3]phpDocumentor is a tool that can be used to generate and parse documentation for PHP. See `https://www.phpdoc.org/about`.

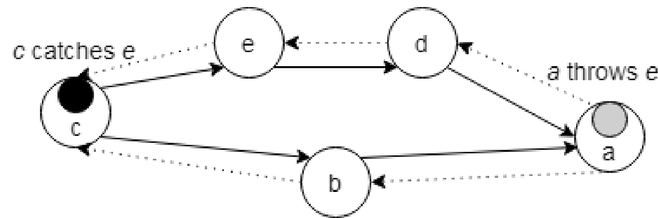



Figure 5.3: Propagation via multiple paths for exception *e* to the catch clause in scope c

**RQ3.** For each each caught exception in the system, we calculate the distance in the class hierarchy between the type of the caught exception and the type by which it was caught. We also record the distance between the caught exception and the exception root type. In Listing 5.4, the exception is caught by subsumption. From the class hierarchy, depicted in Figure 5.5, we can see that the distance between the catch clause and the caught type is 1, and that the distance from the caught exception to the root exception type is 3.

```
1 <?php
2 try {
3   throw new RangeException();
4 } catch (RuntimeException $e) {}
```

Listing 5.4: Catch by subsumption code example

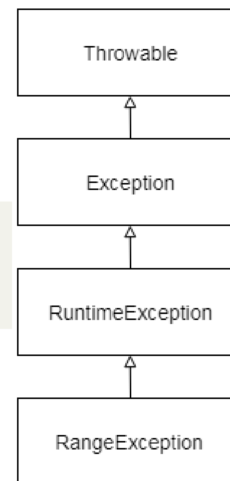



Figure 5.5: Catch by subsumption example class hierarchy

**RQ4.** For answering this question, we use three of the earlier mentioned resources: the exception flow, the `@throws` annotations and the method order. To start with, we record whether an exception that is explicitly raised in a method is documented with a `@throws` annotation. Secondly, we compare the annotations of methods to the exceptions that they encounter but that have their origin in another scope (i.e., they are propagated and/or uncaught). Thereafter, we compare the annotations of abstract methods to the implementations of those methods. The abstract super method defines a “contract” to which the implementations of that method have to adhere to, i.e. they are not allowed to throw exceptions that are not annotated at the abstract super method.

There are three possible outcomes for each exception:

a) It can be annotated, which means that the encountered exception has a corresponding `@throws` annotation;
b) An annotation can be missing, which means that for an encountered exception of type $t$, no corresponding `@throws` annotation with type $t$ could be found;
c) Because phpDocumentor might be unable to parse the type in a `@throws` annotation, the last possible outcome for the question whether an exception is annotated or not is "unknown".

In Chapter 4, we showed that some exceptions could have an "unknown" type. These exceptions are left out the analysis for annotations, as we cannot compare it to an annotation.

## 5.4 Threats to Validity

There are several aspects which threaten the validity of our study. To start with, we limited the scope of our study (see Section 3.4), and the elements which we put out of scope might affect the exception flow. In our evaluation (see Chapter 4), we took some of these features into account. Despite the influence of these features, the aggregated recall and precision concerning the *encounters* set of our approach still were 0.63 and 0.91 respectively (see Section 4.3). The evaluation of our tool gives us confidence that although the outcome of our empirical study might not be completely correct, it will still be close enough to the actual situation to be useful. However, this confidence is based on the assumption that the corpus used for evaluating the approach is representative. The possibility that the corpus is biased poses another threat to the validity. The same holds for the representativeness of the corpus used in the empirical study. We have tried to counter these threats by selecting projects with a wide range of sizes, use cases and backgrounds.

# Chapter 6

# Results

In this chapter we present the results of the empirical study (see Chapter 5). We answer each of the research questions in a separate section.

## 6.1 RQ1. Number of Encounters per Method

The distribution of the number of encounters per method follows a power law distribution. All projects in the corpus yielded similar results in this respect. 70.9% of all the methods in the corpus do not encounter an exception. Of the remaining methods, 21% encounter between 1 and 10 exceptions. 8.1% of the methods in the corpus encounter more than 10 exceptions, with a maximum of 775 exceptions in the `Task::perform` method of Phing.

## 6.2 RQ2. Paths of Exceptions

Figure 6.1 depicts the shortest path lengths from the catch clause where an exception is caught to the initial scope of the caught exception. There is a peak in the graph at length 3, which means that most caught exceptions are raised, propagated to another scope, then propagated to a guarded scope which then catches the exception. Most of the projects in the corpus have a path length distribution which approximates the curve in the aggregated graph. The projects with a lower number of `try`/`catch` and `throw` statements diverge the most from the aggregated curve. Figures for individual projects can be looked up in Appendix A.1.

An important observation is that there exist paths of length 1. This means that an exception is thrown within a guarded scope, and then immediately caught. A second aspect of this curve is that it does not flatten out directly after the peak at length 3: 33.2% of the paths have a length greater than or equal to 5 and smaller than or equal to 10.

## 6.3 RQ3. Catch by Subsumption

Figure 6.2a shows the distances between all caught exceptions in all projects and the type by which those exceptions are caught. A distance of zero means that the exception is caught

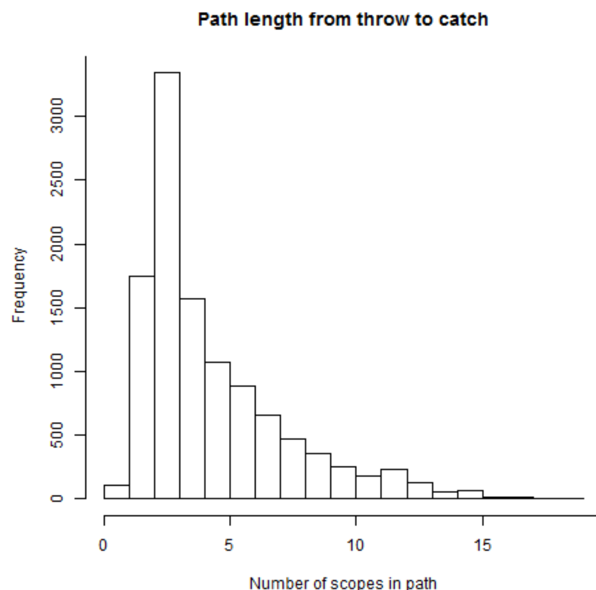



Figure 6.1: Shortest path lengths from `throw` statement to `catch` clause

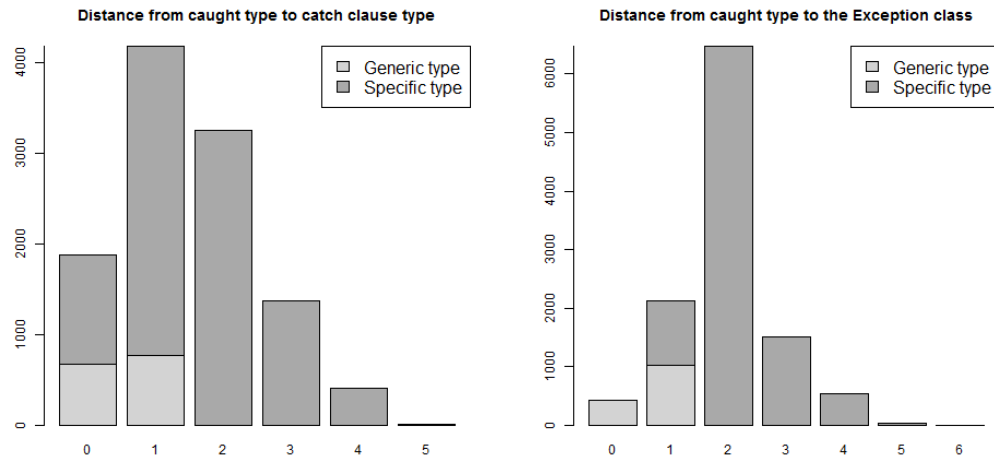
by its direct type. We can see that most exceptions, namely 37.7%, are caught by their direct parent, followed by their parent in the second degree. Figure 6.2b shows the distance from the caught exceptions to the `Exception` type[1]. Most caught exceptions have a distance of two to the `Exception` type.

Both figures also show the usage of 'generic' exception types versus 'specific' exception types. We consider every occurrence of `Exception` or `RuntimeException` as 'generic', as it does not signal what kind of exception did occur. We categorise all other exceptions as 'specific'. If we combine the knowledge of the two graphs, we can infer that most catch clauses use a subtype of `Exception` to catch an exception that is their direct child. However, the data from individual projects can in some cases diverge strongly from the aggregated data; this aggregate is dominated by the Joomla project, which accounts for 3090 of the 4190 exceptions caught by their direct parent as presented in Figure 6.2a. In Appendix A.2 we present Figures for both the distance between the caught types and the type by which they are caught, as well as for the distance between the caught types and the `Exception` type for all projects in the corpus.

A project which strongly deviates from the aggregate is PHPUnit. In PHPUnit, exceptions are most often caught by their direct type, as can be seen in Figure 6.3a. In addition, PHPUnit never uses an exception type which we categorised as generic. Figure 6.3b shows that PHPUnit only uses exception types with a minimum distance of two to the `Exception`

[1] In PHP versions before version 7.0, `Exception` is the root type of the exception class hierarchy. Starting from version 7.0, the `Throwable` interface was added, which is implemented by `Exception`. Because most projects in our corpus have a minimum required version of PHP 5.x, we decided to assume that the `Exception` class is the root for the exception hierarchy in all projects.

type.




(a) Distances from all caught exceptions to the types of the catching catch clause in all projects

(b) Distance from the caught exceptions to the `Exception` type in all projects

Figure 6.2: Distances between exceptions, catch clauses and root types in all projects

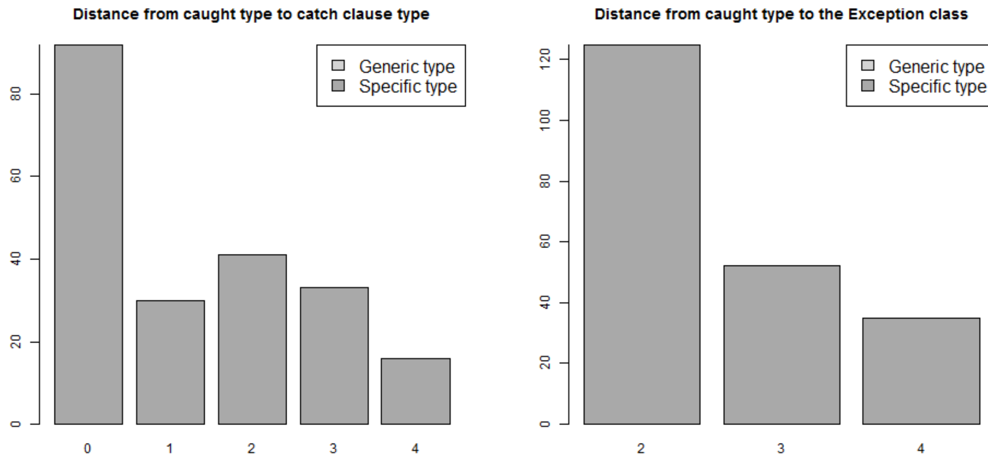
## 6.4 RQ4. Documentation Validation

Figure 6.4 depicts the presence of annotations for encountered exceptions. Figure 6.4a focuses on exceptions that are explicitly thrown in a method. Figure 6.4b shows the exceptions that are encountered in a method but not explicitly thrown compared to the annotations of that method. Lastly, Figure 6.4c shows the annotations at contract level, compared to the encountered exceptions in its implementations. For each project in the corpus, similar figures are presented in Appendix A.3.

Figure 6.4a shows that most explicitly thrown exceptions are not annotated. Even if all annotations for which we could not infer an exception type would be annotated, there still would be more exceptions which are not annotated. This does not hold for all projects: CakePHP, Joomla, Phing, and Smarty have more than 100 `throw` statements, and all have more annotated than not annotated exceptions[2].

For propagated and uncaught exceptions, there are even fewer annotations, as can be seen in Figure 6.4b. Here, Smarty and Roundcube have a larger portion of annotated exceptions as compared to the other projects in the corpus.

[2]Note that not every `throw` statement should lead to a `throws` annotation, e.g. if the exception is directly handled in the method in which it is thrown.



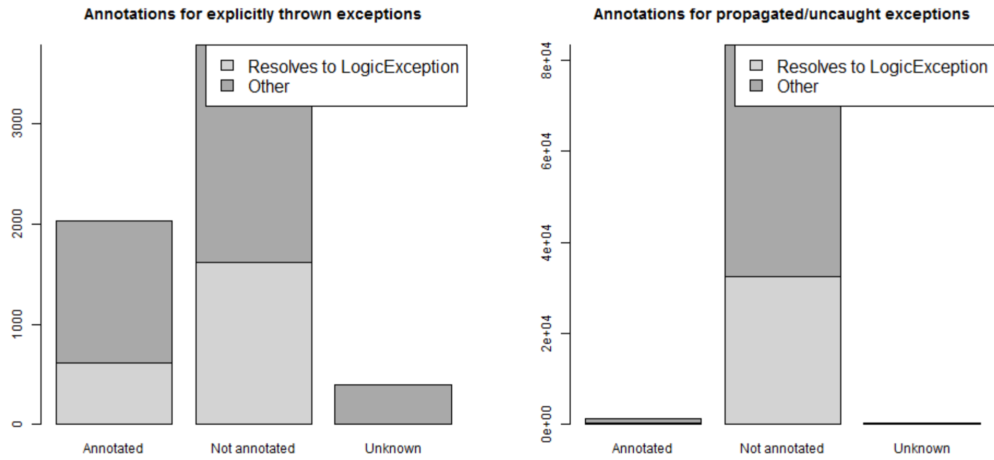

(a) Distances from all caught exceptions to the types of the catching catch clause in PHPUnit

(b) Distance from the caught exceptions to the `Exception` type in PHPUnit

Figure 6.3: Distances between exceptions, catch clauses and root types in PHPUnit

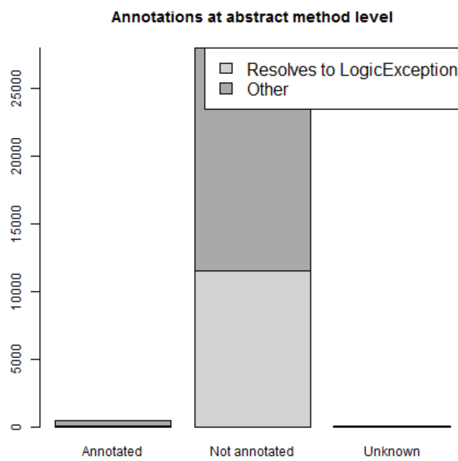
Figure 6.4c shows the encountered exceptions in methods compared to the annotations at abstract super method level. We can see that the vast majority of encountered exceptions is not annotated at abstract method level. It is worth noting that some projects (i.e. CakePHP, phpDocumentor, PHPUnit, Roundcube, Wordpress) do not have any exceptions documented at abstract method level.

In each of these figures, we separated the exceptions that can be resolved to `LogicException`. The PHP manual states that a `LogicException` "should lead directly to a fix in your code" [4]. In other words, a `LogicException` does not have to be handled or documented, as it should not be caught but "fixed". Following this line of reasoning, we decided to separate `LogicExceptions` from the other exceptions to see whether this is also done in practice. When looking at the the graphs in Figure 6.4, we see that there is not a big difference between "normal" exceptions and `LogicExceptions` in terms of annotations. 34.9% of the normal exceptions are annotated if they are thrown directly, whereas 27.3% of the `LogicExceptions` are annotated. For propagated and uncaught exceptions it holds that 1.4% of the normal exceptions are annotated, versus 0.7% of the `LogicExceptions`. Normal exceptions are annotated at interface level in 1.6% of the cases, compared to 0.6% of the `LogicExceptions`.




(a) Annotations compared to explicitly raised exceptions in all projects

(b) Annotations compared to encountered (but not raised) exceptions in all projects




(c) Annotations at abstract method level compared to encountered exceptions in implementing methods in all projects

Figure 6.4: Comparison of `throws` annoations to encountered exceptions

# Chapter 7

# Discussion

In this chapter, we will discuss the results as presented in Chapter 6. Thereby, we answer the research questions we raised in Chapter 5. In Section 7.5, we discuss the implications of the accuracy of the algorithm.

## 7.1 RQ1. Number of Encounters per Method

Most methods do not encounter any exception at all: in only 30% of all methods, an exception can occur. Most methods in which exceptions can come acoss only encounter a small number of exceptions. We believe that it is unlikely that a programmer who writes a method that encounters many exceptions (e.g. 15 or more) is aware of all of them. It would therefore be interesting to see whether these "unnoticed" exceptions cause bugs, which could be researched in the future. If this is the case, our tool could be used to find methods that do encounter many exceptions, so that these methods can be refactored. Another subject which we left untouched is concerned with the nature of methods that do encounter many exceptions. A qualitative study of those methods might point out similarities between those methods.

## 7.2 RQ2. Paths of Exceptions

As mentioned in Section 5.1, it is considered a good practice to catch an exception as close to the source as possible. Although this criterion is hard to express in numbers, we deem it acceptable if an exception travels through 2 upto 4 scopes before it is caught. Using this criterion, 69.4% of the exceptions is actually caught 'close to the source'.

29.7% of the caught exceptions has been in more than five scopes before they are caught. We can think of several explanations for this fact:

1. The caught exception originated from a dependency and not from the project in which it was caught itself. In this case, the exception has traveled through a number of framework scopes before it ends up in the project itself. We leave this question open for future research.

2. The caught exception was unknown to the developer. We think that a developer who is writing a catch clause is often not aware of exceptions that come from functions that are 5 or more calls away. This leads to the conclusion that the developer who wrote a catch clause that catches such an exception probably did not intend to catch it. This problem can be prevented by using specific exception types for both throwing and catching, as it is less likely to catch an exception unintentionally if the catch clause can only catch a small subset of exception types.

3. The `try`/`catch` construct was added as a safeguard. In some instances, a developer might not know whether the method they are using throws or propagates any exceptions. For example, this can be the case when the method resides in another subsystem of the project, or if it is an external method. Because the developer does not want the method containing the call to that untrusted method to propagate an exception, a `try`/`catch` block is added with a generic type in the catch clause.

Another result that requires attention is the fact that there are cases in which an exception is caught in the same scope as it was raised. These cases only make up for 0.9% of the paths. The exception can in these cases be reduced to a `goto` statement, as the throw statement is used to jump over a set of instructions to the instructions in the catch clause. Dijkstra [15] argues that the `goto` statement should be considered as harmful. Therefore, we argue that cases where an exception is thrown and immediately caught should be refactored.

## 7.3 RQ3. Catch by Subsumption

In the optimal situation, the graph showing the distance between a caught type and the type by which it is caught is right skewed: this indicates that the distance between the caught type and the type in the catch clause is small most of the times. From Figure 6.2a, we know that this is not the case. This fact can be interpreted in multiple ways. First, it is possible that the programmer who wrote the catch clause did not care about the precision. It could be that they simply wanted to log the exception, but not handle it. In other words, the writer of the catch clause does often not seem to care about the precision that is offered by the specific exception type. Conversely, we can say that the programmer who is writing the function that throws the exception offers information which is lost because the exception is caught with a type that is too generic. In some cases, the caught exception might have been handled correctly, but this is prevented by using the wrong catch clause.

In Figure 6.2b, we can see that most exceptions do have a specific type. Although the exceptions that are caught do have specific types most of the times, the catch clauses do often not use this specific information. A possible explanation is that PHP programmers use generic types in catch clauses because they might not be certain about which exception types might occur in a certain piece of code. They then use a generic type so that no exception escapes.

## 7.4 RQ4. Documentation Validation

For 56.4% of the explicitly thrown exceptions, no annotation was available. When subtracting all exceptions that resolve to `LogicException`, following the reasoning that these do not have to be annotated, still 51.4% of the thrown exceptions would not be documented. Exceptions which do not originate from the method scope are even less often documented: 98.3% of the propagated exceptions are not documented when incorporating `LogicException`s, and 98% when not accounting for `LogicException`s. As we can see, this large percentage cannot be explained by the fact that `LogicException`s do not have to be documented. We can think of several explanations for this fact, each of which could be researched in future work.

- Exceptions thrown in private methods might be annotated less often. This might happen because the private method can by definition only be used locally. The developer who wrote the private method is more likely to be the only one who uses that method than if it were a public method. The method is not part of the public interface and is thus not a method of which the documentation has large impact.

- Exceptions thrown in short methods might be annotated less often. The reasoning behind this explanation is that an exception which is thrown in a method of less than 10 lines is often easy to read. The `throw` statement is more likely to be noticed in such a method, and therefore, the developer does not deem it neccessary to annotate the thrown exception.

- Exceptions might not be annotated because of laziness. Although creating such an annotation is not hard, it requires some extra work which does not yield an immediate reward.

Exceptions were in 98.4% of the all the cases not annotated at interface level. If we consider only the exceptions that do not resolve to `LogicException`, still 97.9% of all exceptions were not annotated at interface level. We did not discern between explicitly thrown and propagated exceptions. We consider each exception that is thrown in an implementing class but not documented in the interface as a violation of the Liskov Substitution Principle. Liskov and Wing [22] define the "exception rule" as follows: "The exceptions signaled by $m_\sigma$, are contained in the set of exceptions signaled by $m_\rho$." Here, method $m_\sigma$ overrides $m_\rho$. If we interpret the word "signaled" as "annotated", we can see that each exception that is propagated by a method in a subtype and not annotated in the parent of that method is a violation of this rule.

As we can see, this violation is abundant in PHP programs. We are interested in whether these violations do have a big impact and pose this as a question for future research. The same holds for the consequences of non-annotated exceptions in methods that propagate or throw exceptions themselves.

## 7.5 Algorithm Accuracy

In Chapter 4, we have determined the accuracy of the algorithm by comparing the output of the algorithm to three manually created oracles. The precision is 91%, and almost all exceptions that were in the output but not in the oracle had a very generic type (e.g. "object"). These generic types can be filtered out the output of the algorithm easily, and thus have a low impact. The impact of the incompleteness of the algorithm is more serious, as 37% of the exceptions is not recalled by the algorithm. As most of the results of our empirical study show a clear trend, we believe that the recall of our algorithm is sufficient to justify our conclusions concerning those results.

The accuracy of the algorithm can be improved in several ways. First, improving the type inference algorithm will result in an improved accuracy. As a starting point, better support for resolving types of class properties could be implemented. Another possible improvement could be made by implementing support for dynamic calls. Lastly, implementing correct support for `try`/`catch` constructs in the CFG creation algorithm will also improve the accuracy of the algorithm.

# Chapter 8

# Related Work

Modeling the exception flow of a system has been done in multiple ways. We discuss the literature on exception flow modeling in Section 8.1. There are also numerous studies available which are concerned with exception and error handling. In Section 8.2, we present a subset of these studies. PHP is a less well explored topic in scientific literature, but some of the existing work on PHP is also related to our work. We introduce the most important works on PHP and web development with relation to our work in Section 8.3.

## 8.1 Exception Flow Modeling

Amighi et al. [7] created an algorithm which models the control flow graph of software systems with focus on exceptions. Their approach uses an intermediate representation of the code to be analysed, in contrast to the approach by Robillard and Murphy [26] which is based on analysing the AST. Their approach is specifically designed for the Java programming language. Sinha and Harrold [27] analysed the impact of exceptions on static analysis tools. They also constructed an algorithm for exception propagation in Java, which uses CFGs for intraprocedural exception flows. To be able to model the interprocedural exception flow, they link the CFGs together into an (interprocedural) CFG. In contrast to the PHP-CFG library, their CFG construction algorithm correctly accounts for `try`/`catch` constructs. Fu and Ryder [17] have created an algorithm which is able to compute semantically related exception chains. Exception paths are linked in a chain if one of the following conditions applies: either an exception is caught and rethrown, or a new exception which is based on a caught exception is thrown. Their work could be used to expand our own analysis.

## 8.2 Exception and Error Handling

Coelho et al. [12] conducted an extensive study on exception handling bug hazards in Android projects. In their study, they analysed over 6000 stack traces and surveyed 71 developers. They found that most errors in Android projects are due to errors in programming logic. In addition, they found that many unchecked exceptions are also undocumented, which is

in concordance with our own finding concerning exception documentation in PHP projects. Ebert et al. [16] executed a survey of 154 developers and manually analysed 220 exception handling bugs from two Java projects. From the results of their survey, they found that most developers do not test the exception handling constructs in their code. They also found that exception handling bugs did only form a small portion of the complete set of bugs in the analysed projects. Marinescu [23] did an empirical study of three releases of the Eclipse project. In this study, they found that classes that do use exceptions are more likely to be error prone. This finding supports the notion that tooling that helps programmers to understand exception handling is needed. Cabral and Marques [10] studied 32 .NET and Java projects to find how these projects deal with exception handling. They found that exception handling constructs are often not used correctly, and pose the question whether the design of exception handling mechanisms might be flawed.

## 8.3 PHP and Web Development

Hills [20] has created a collection of variable feature usage patterns and anti-patterns in PHP. They analyse how often these patterns occur in 20 open-source projects, and show how often these patterns can be statically resolved. They conclude that these patterns can be used to improve the accuracy of static analysis tools. We believe the implementation of these patterns in the type inference algorithm will increase the performance of our tool.

Nguyen et al. [25] introduce WebSlice, a tool for slicing web applications. WebSlice is able to create cross-language slices as web projects often consist of both PHP, HTML, JavaScript and SQL code. They show that many program slices do indeed cross languages, indicating the complexity of web application development. Their approach could be used to create a slicer which also accounts for the exception flow. Such a slicer would make it possible to see the impact of an uncaught exception on the web application as a whole.

# Chapter 9

# Conclusion

In this chapter, we reflect on the contents of our research. To start with, we list our contributions in Section 9.1. Thereafter, we summarise our findings in Section 9.2. Lastly, we provide some pointers for future work in Section 9.3.

## 9.1 Contributions

Our work contains three main contributions. The first contribution is our implementation of the exception flow detection algorithm of Robillard and Murphy [26] for the PHP language. We have shown that it is possible to detect the exception flow in dynamically typed languages, by first executing a type inference algorithm. Our second contribution is the determination of the accuracy of our implementation of the algorithm of Robillard and Murphy [26]. In the original paper, the authors only presented a few use cases in which their algorithm would be useful, but they did not show how accurate their algorithm is. The third main contribution is the empirical study we performed on 20 PHP projects, focusing on their exception flow.

## 9.2 Conclusions

We focus our conlusions on two parts of our thesis: the accuracy of the algorithm, and the results of our empirical study. To start with, the evaluation of our approach showed that the recall of the algorithm is 0.63, meaning that of all exceptions that could have been encountered, our approach detects 63%. Of the output of our approach, 91% of the exceptions are exceptions that can be actually encountered. Improvement in the underlying type inference algorithm will also lead to an improvement of the accuracy of our approach.

The results of our empirical study provide insight into the way PHP programmers use exceptions. We have shown that most methods do not encounter any exceptions at all. We have also shown that most exceptions are caught by subsumption, i.e. they are not caught by their exact type. This signals that developers often do not make use of the extra information that is provided by the specific exception type. Exceptions are most of the time caught after they have traveled through three scopes. However, paths with lengths of up to ten are quite

common. These long paths might signal a problem, as the developer that wrote a catch clause might not be aware of all the exceptions that are caught. We have also shown that PHP developers catch most exceptions by subsumption. Lastly, we compared the `@throws` annotations of methods to the exceptions that are actually encounted. We found that most exceptions are not annotated, even if they are thrown explicitly. For methods which override an abstract method, we also compared the annotations of the abstract method to the encountered exceptions of the overriding method. We found that many methods violate the Liskov Substitution Principle by propagating exceptions which are not documented at the abstract method.

## 9.3 Future Work

We can think of several opportunities for future work. A first suggestion for future work is improving the type inference algorithm. During the evaluation of our approach, we found that the type inference algorithm is of great importance to the accuracy of our approach. Better support for properties and for the dynamic features of PHP would improve the accuracy of our algorithm. A different opportunity for improving the accuracy of our approach lies in implementing proper support for `try`/`catch` constructs in the PHP-CFG library.

Another research subject could be finding whether developers that write catch clauses know or even want to know about all the exceptions that are caught these catch clauses. We found that some catch clauses catch exceptions which have traveled through many scopes before they are caught, and wonder whether this is intentional and whether it could be harmful.

Our last suggestion for future work is researching the impact of exceptions which are encountered but not annotated. A future study could try to find why developers do not annotate exceptions, and if exceptions that are not annotated to fatal errors lead more often.

# Bibliography


[1] Google trends - laravel, symfony, codeigniter, yii, cakephp. https://trends.google.nl/trends/explore?q=laravel,symfony,codeigniter,yii,cakephp. Accessed: 2017-04-25.

[2] monolog/monolog. https://packagist.org/packages/monolog/monolog. Accessed: 2017-04-25.

[3] PHP: Exceptions - Manual. http://php.net/manual/en/language.exceptions.php, . Accessed: 2017-08-15.

[4] PHP: LogicException - Manual. http://php.net/manual/en/class.logicexception.php, . Accessed: 2017-08-03.

[5] The best PHP framework for 2015: Sitepoint survey results. https://www.sitepoint.com/best-php-framework-2015-sitepoint-survey-results/. Accessed: 2017-04-25.

[6] TIOBE index for june 2017. https://www.tiobe.com/tiobe-index/. Accessed: 2017-06-07.

[7] A. Amighi, Pedro de Carvalho Gomes, and Marieke Huisman. *Provably Correct Control-Flow Graphs from Java Programs with Exceptions*, pages 31–48. Number 26 in Karlsruhe Reports in Informatics. Karlsruhe Institute of Technology, 10 2011.

[8] E. A. Barbosa, A. Garcia, and S. D. J. Barbosa. Categorizing faults in exception handling: A study of open source projects. In *2014 Brazilian Symposium on Software Engineering*, pages 11–20, Sept 2014.

[9] Matthias Braun, Sebastian Buchwald, Sebastian Hack, Roland Leißa, Christoph Mallon, and Andreas Zwinkau. Simple and efficient construction of static single assignment form. In *Proceedings of the 22Nd International Conference on Compiler Construction*, CC'13, pages 102–122, Berlin, Heidelberg, 2013. Springer-Verlag.

[10] Bruno Cabral and Paulo Marques. Exception handling: A field study in Java and .NET. In *Proceedings of the 21st European Conference on Object-Oriented Programming*, ECOOP'07, pages 151–175, Berlin, Heidelberg, 2007. Springer-Verlag.

[11] Byeong-Mo Chang, Jang-Wu Jo, and Soon Hee Her. Visualization of exception propagation for Java using static analysis. In *Proceedings of the Second IEEE International Workshop on Source Code Analysis and Manipulation*, SCAM '02, pages 173–182, Washington, DC, USA, 2002. IEEE Computer Society.

[12] Roberta Coelho, Lucas Almeida, Georgios Gousios, Arie van Deursen, and Christoph Treude. Exception handling bug hazards in Android. *Empirical Software Engineering*, 22(3):1264–1304, Jun 2017.

[13] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of the 9th European Conference on Object-Oriented Programming*, ECOOP '95, pages 77–101, London, UK, UK, 1995. Springer-Verlag.

[14] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numer. Math.*, 1 (1):269–271, December 1959. ISSN 0029-599X.

[15] Edsger W. Dijkstra. Letters to the editor: Go to statement considered harmful. *Commun. ACM*, 11(3):147–148, March 1968. ISSN 0001-0782.

[16] Felipe Ebert, Fernando Castor, and Alexander Serebrenik. An exploratory study on exception handling bugs in Java programs. *Journal of Systems and Software*, 106: 82–101, 2015.

[17] Chen Fu and Barbara G. Ryder. Exception-chain analysis: Revealing exception handling architecture in Java server applications. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 230–239, Washington, DC, USA, 2007. IEEE Computer Society.

[18] I. Garcia and N. Cacho. eFlowMining: An exception-flow analysis tool for .NET applications. In *2011 Fifth Latin-American Symposium on Dependable Computing Workshops*, pages 1–8, April 2011.

[19] John B. Goodenough. Exception handling: Issues and a proposed notation. *Commun. ACM*, 18(12):683–696, December 1975. ISSN 0001-0782.

[20] M. Hills. Variable feature usage patterns in PHP (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 563–573, Nov 2015.

[21] Etienne Kneuss, Philippe Suter, and Viktor Kuncak. On using static analysis to detect type errors in PHP applications. Technical report, Ecole polytechnique fédérale de Lausanne, 2010.

[22] Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6):1811–1841, November 1994. ISSN 0164-0925.

[23] Cristina Marinescu. Are the classes that use exceptions defect prone? In *Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th Annual ERCIM Workshop on Software Evolution*, IWPSE-EVOL '11, pages 56–60, New York, NY, USA, 2011. ACM.

[24] Robert Miller and Anand Tripathi. *Issues with exception handling in object-oriented systems*, pages 85–103. Springer Berlin Heidelberg, Berlin, Heidelberg, 1997.

[25] Hung Viet Nguyen, Christian Kästner, and Tien N. Nguyen. Cross-language program slicing for dynamic web applications. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 369–380, New York, NY, USA, 2015. ACM.

[26] Martin P. Robillard and Gail C. Murphy. Static analysis to support the evolution of exception structure in object-oriented systems. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 12(2):191–221, 2003.

[27] Saurabh Sinha and Mary Jean Harrold. Analysis and testing of programs with exception handling constructs. *IEEE Transactions on Software Engineering*, 26(9):849–871, 2000.

[28] Henk Erik Van der Hoek and Jurriaan Hage. Object-sensitive type analysis of PHP. In *Proceedings of the 2015 Workshop on Partial Evaluation and Program Manipulation*, PEPM '15, pages 9–20, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3297-2.

[29] Merijn Wijngaard. Dependence analysis in PHP. Master's thesis, Universiteit van Amsterdam, The Netherlands, 2016.

[30] Rebecca J. Wirfs-Brock. Toward exception-handling best practices and patterns. *IEEE software*, 23(5):11–13, 2006.

# Appendix A

# Results

Chapter 6 depicts mainly aggregated data. This appendix contains all relevant figures for each project in the corpus, similar to the figures in Chapter 6.

## A.1 Paths of Exceptions

This section contains Figures depicting the lengths of the shortest paths for caught exceptions from `throw` statement to `catch` clause for the projects in the corpus. If a project does not catch any exceptions, it has not been included in this overview.

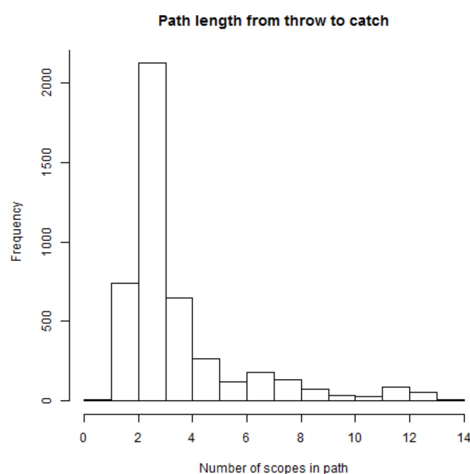



Figure A.1: Shortest path lengths from `throw` statement to `catch` clause in Joomla

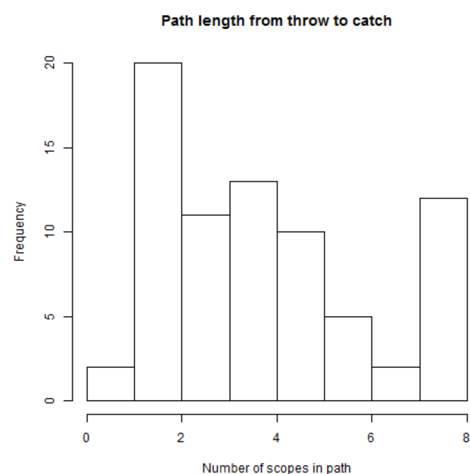



Figure A.2: Shortest path lengths from `throw` statement to `catch` clause in CakePHP

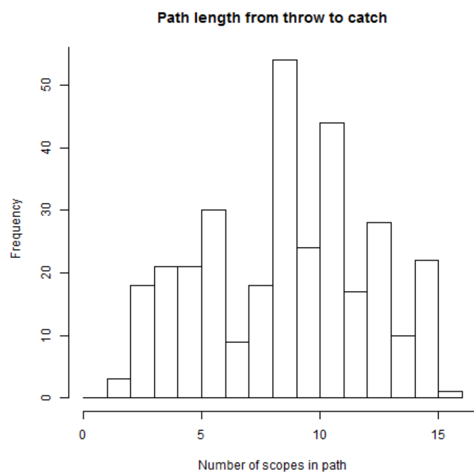



Figure A.3: Shortest path lengths from `throw` statement to `catch` clause in Doctrine

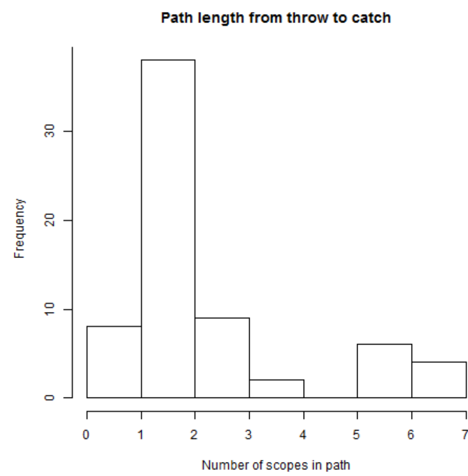



Figure A.4: Shortest path lengths from `throw` statement to `catch` clause in DokuWiki

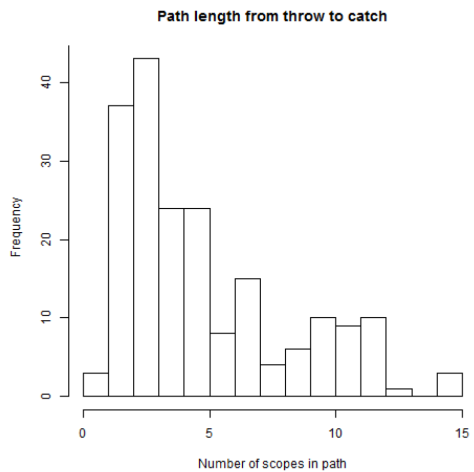



Figure A.5: Shortest path lengths from `throw` statement to `catch` clause in Drupal

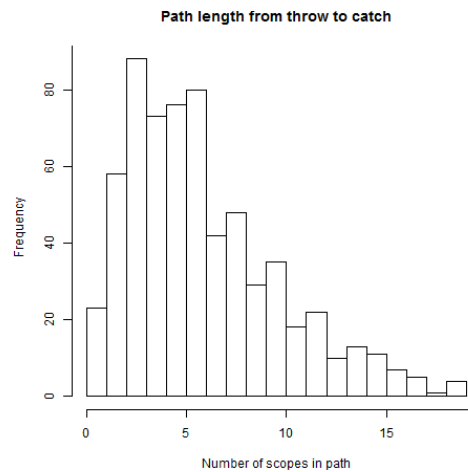



Figure A.6: Shortest path lengths from `throw` statement to `catch` clause in Fabric

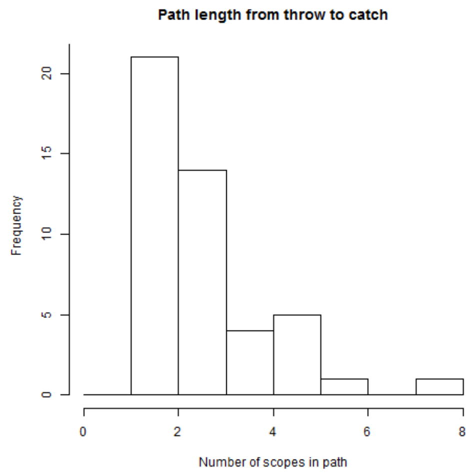



Figure A.7: Shortest path lengths from `throw` statement to `catch` clause in Nette

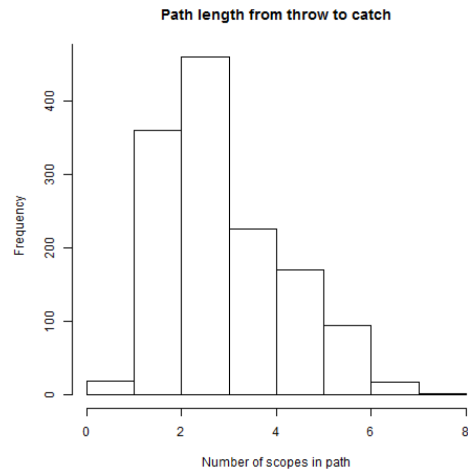



Figure A.8: Shortest path lengths from `throw` statement to `catch` clause in Phing

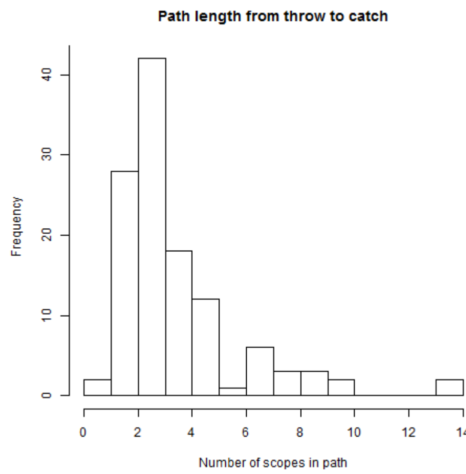



Figure A.9: Shortest path lengths from `throw` statement to `catch` clause in phpBB

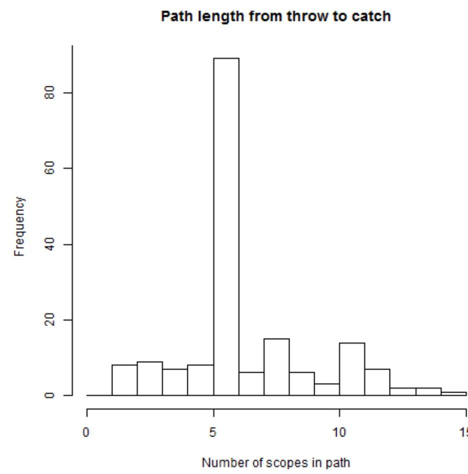



Figure A.10: Shortest path lengths from `throw` statement to `catch` clause in phpDocumentor2

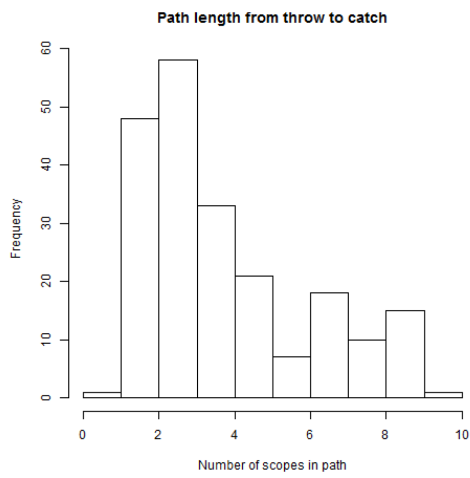



Figure A.11: Shortest path lengths from `throw` statement to `catch` clause in PHPUnit

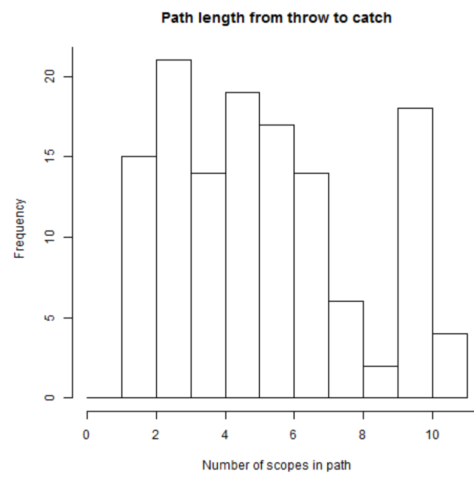



Figure A.12: Shortest path lengths from `throw` statement to `catch` clause in Digitaalloket

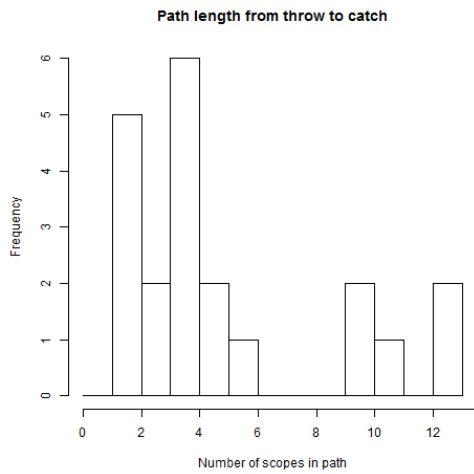



Figure A.13: Shortest path lengths from `throw` statement to `catch` clause in Objectbrowser

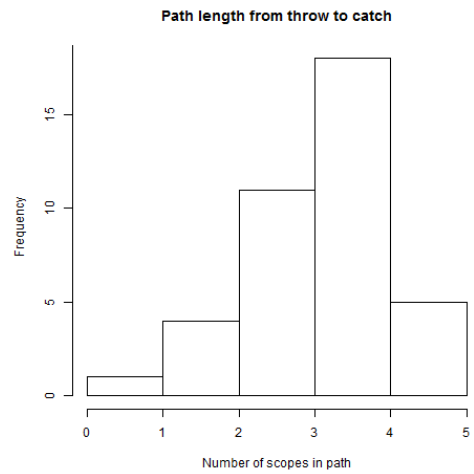



Figure A.14: Shortest path lengths from `throw` statement to `catch` clause in Roundcube

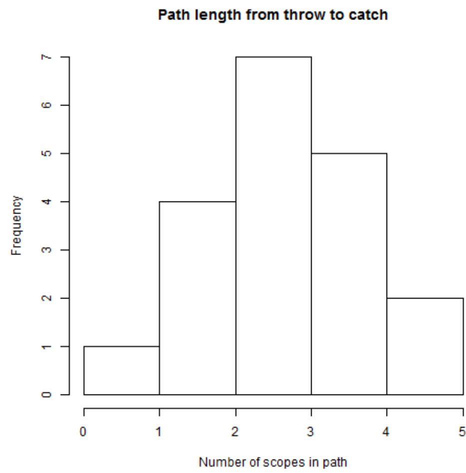



Figure A.15: Shortest path lengths from `throw` statement to `catch` clause in Smarty

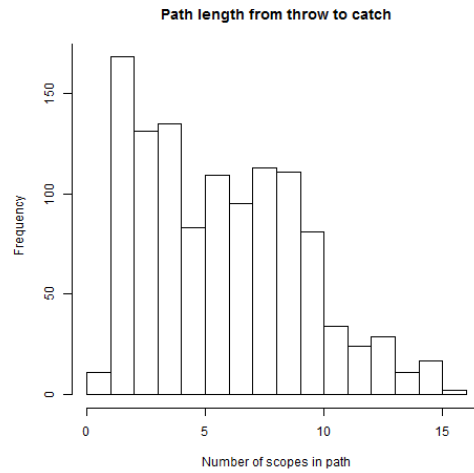



Figure A.16: Shortest path lengths from `throw` statement to `catch` clause in Symfony

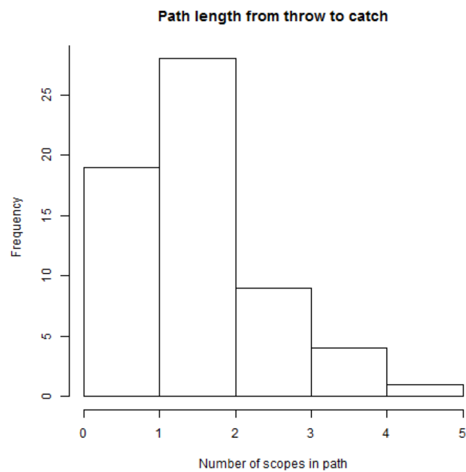



Figure A.17: Shortest path lengths from `throw` statement to `catch` clause in Wordpress

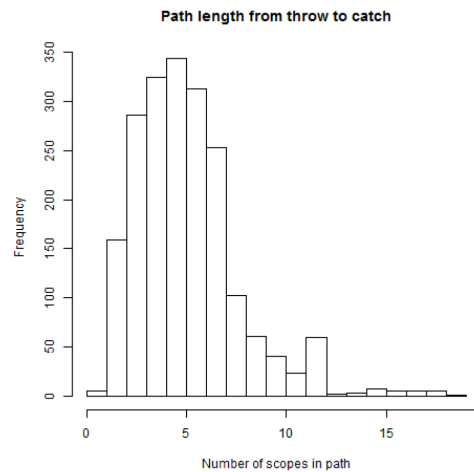



Figure A.18: Shortest path lengths from `throw` statement to `catch` clause in Zend Framework

## A.2 Catch by Subsumption

Section A.2.1 contains Figures depicting the distances between the types of caught exceptions and the type by which they are caught. Section A.2.2 shows Figures which depict the distances between the types of caught exceptions and the `Exception` type.

### A.2.1 Caught Type to Catch Clause Type

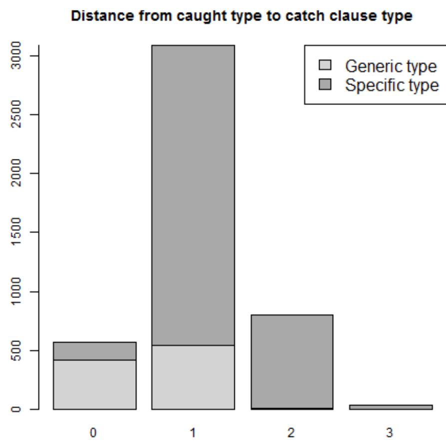



Figure A.19: Distances from all caught exceptions to the types of the catching catch clause in Joomla

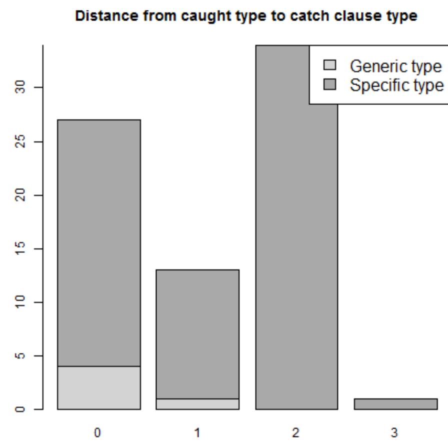



Figure A.20: Distances from all caught exceptions to the types of the catching catch clause in CakePHP

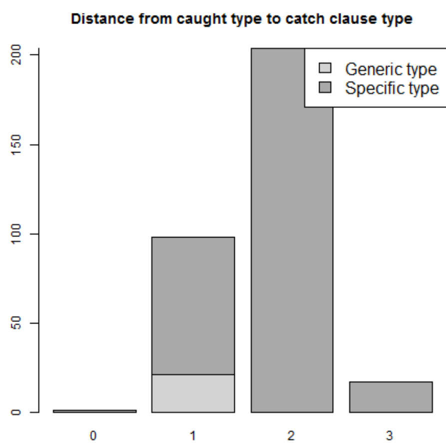



Figure A.21: Distances from all caught exceptions to the types of the catching catch clause in Doctrine

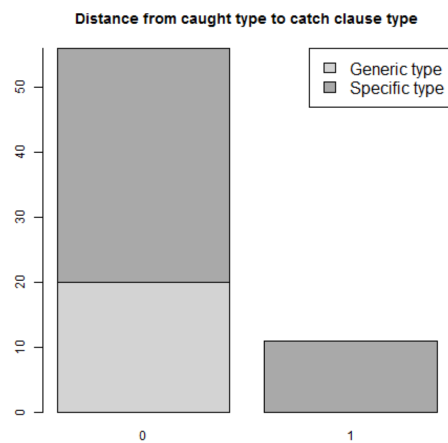



Figure A.22: Distances from all caught exceptions to the types of the catching catch clause in DokuWiki

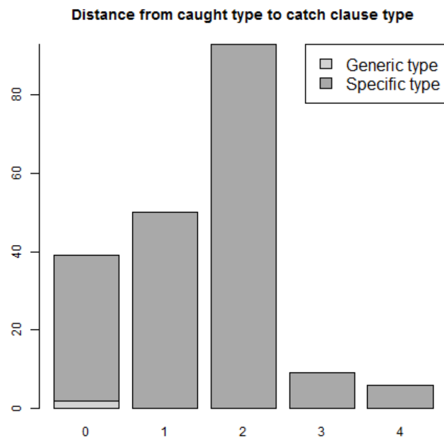



Figure A.23: Distances from all caught exceptions to the types of the catching catch clause in Drupal

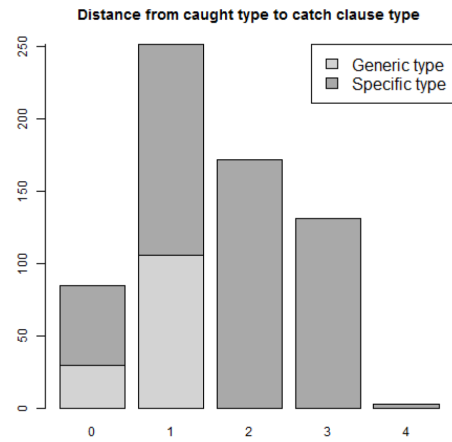



Figure A.24: Distances from all caught exceptions to the types of the catching catch clause in Fabric

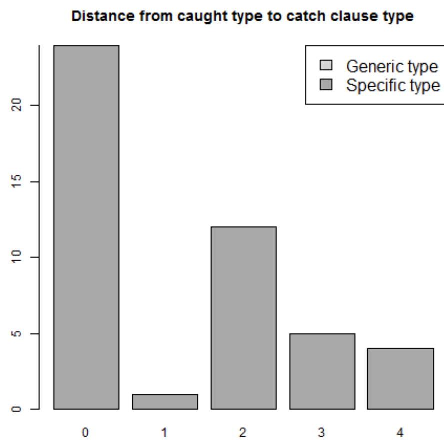



Figure A.25: Distances from all caught exceptions to the types of the catching catch clause in Nette

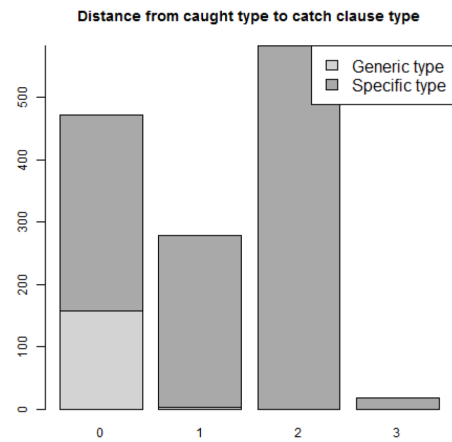



Figure A.26: Distances from all caught exceptions to the types of the catching catch clause in Phing

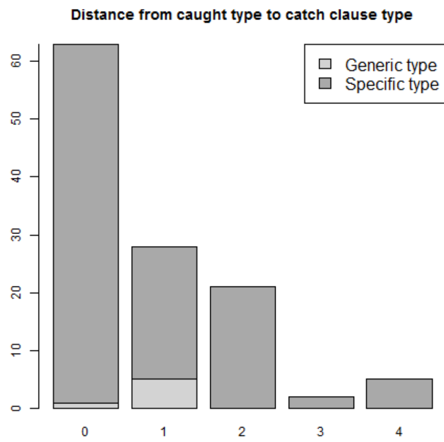



Figure A.27: Distances from all caught exceptions to the types of the catching catch clause in phpBB

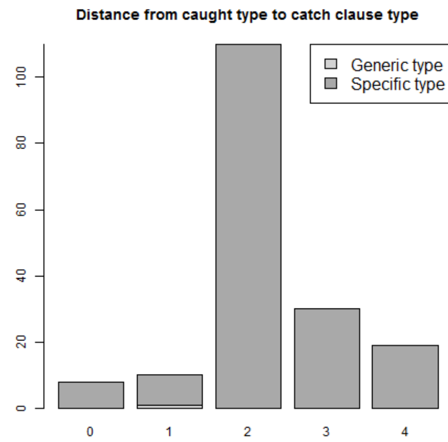



Figure A.28: Distances from all caught exceptions to the types of the catching catch clause in phpDocumentor2

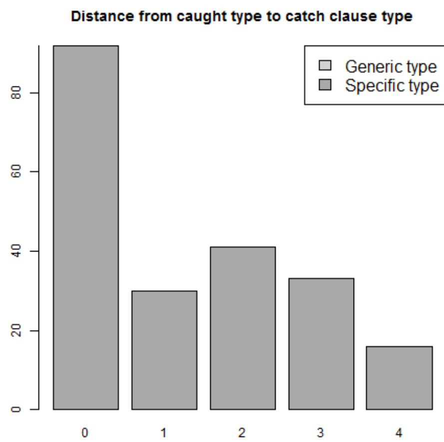



Figure A.29: Distances from all caught exceptions to the types of the catching catch clause in PHPUnit

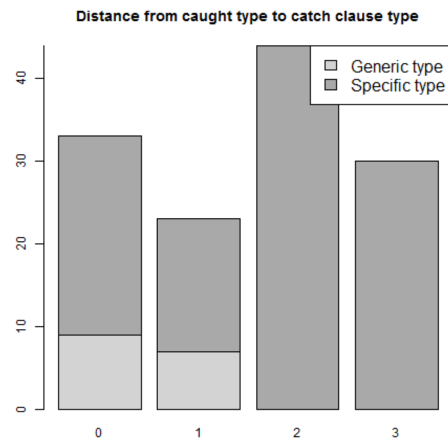



Figure A.30: Distances from all caught exceptions to the types of the catching catch clause in Digitaalloket

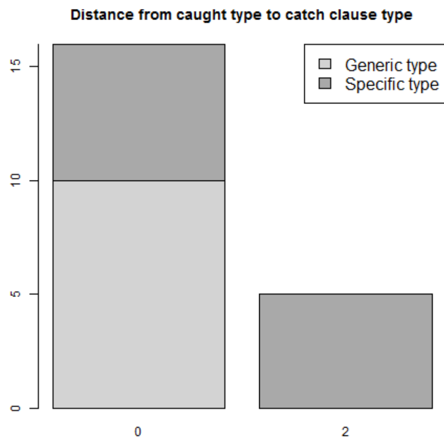



Figure A.31: Distances from all caught exceptions to the types of the catching catch clause in Objectbrowser

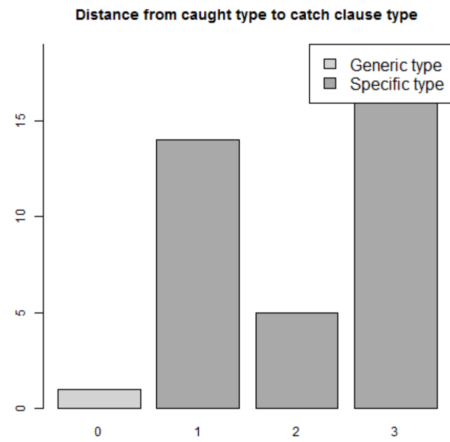



Figure A.32: Distances from all caught exceptions to the types of the catching catch clause in Roundcube

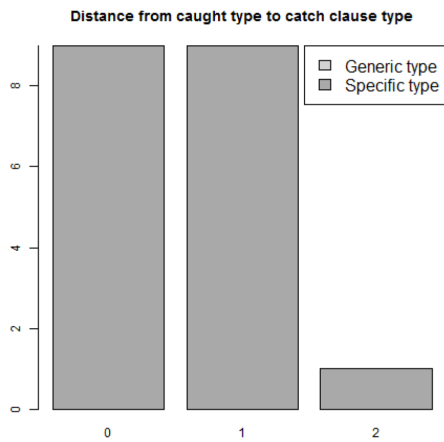



Figure A.33: Distances from all caught exceptions to the types of the catching catch clause in Smarty

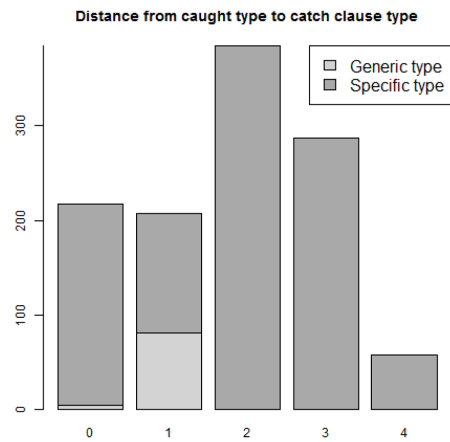



Figure A.34: Distances from all caught exceptions to the types of the catching catch clause in Symfony

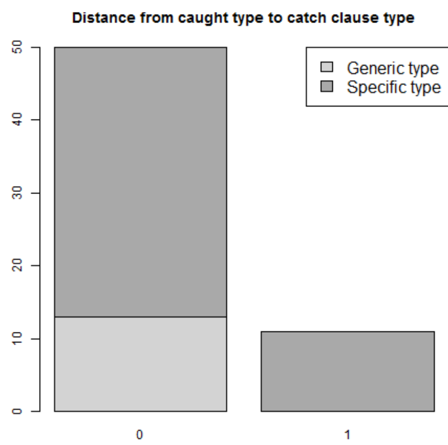



Figure A.35: Distances from all caught exceptions to the types of the catching catch clause in Wordpress

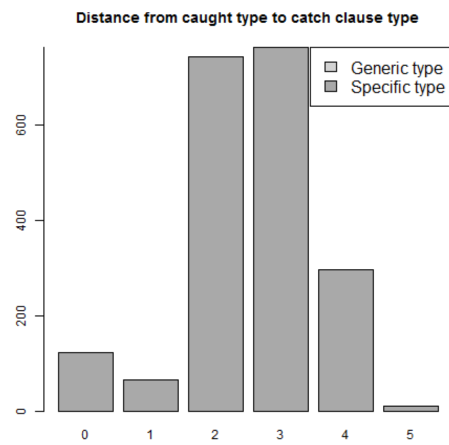



Figure A.36: Distances from all caught exceptions to the types of the catching catch clause in Zend Framework

### A.2.2 Caught Type to `Exception`

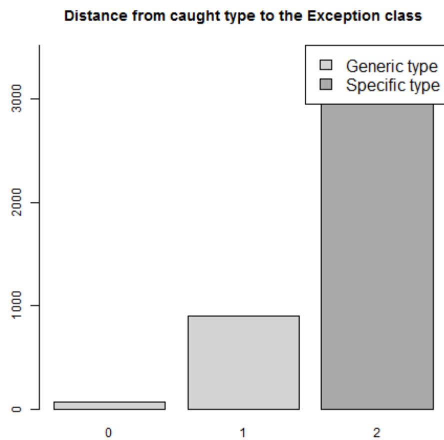



Figure A.37: Distance from the caught exceptions to the `Exception` type in Joomla

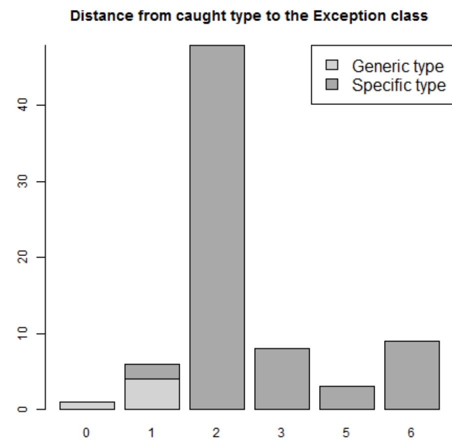



Figure A.38: Distance from the caught exceptions to the `Exception` type in CakePHP

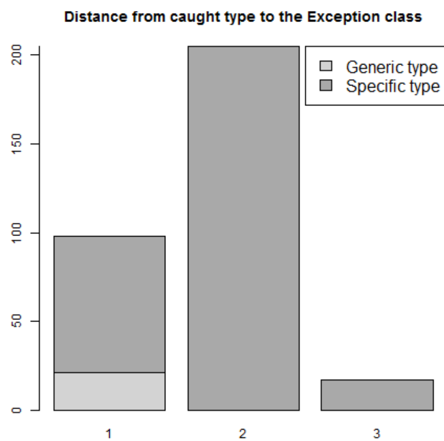



Figure A.39: Distance from the caught exceptions to the `Exception` type in Doctrine

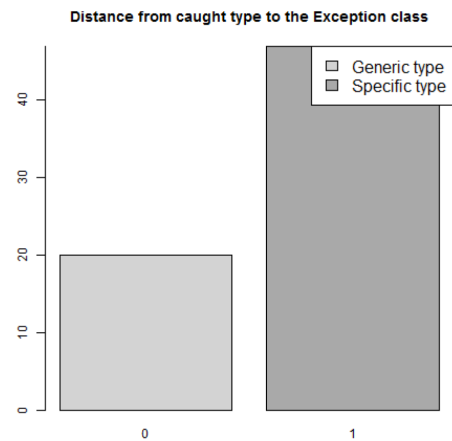



Figure A.40: Distance from the caught exceptions to the `Exception` type in DokuWiki

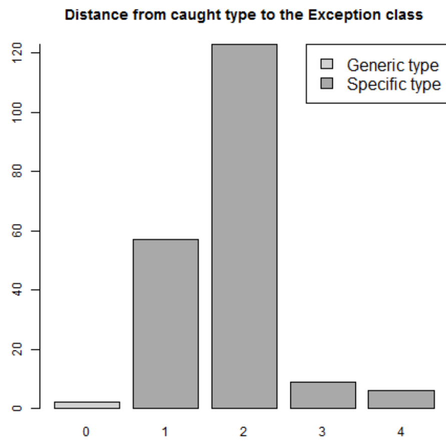



Figure A.41: Distance from the caught exceptions to the `Exception` type in Drupal

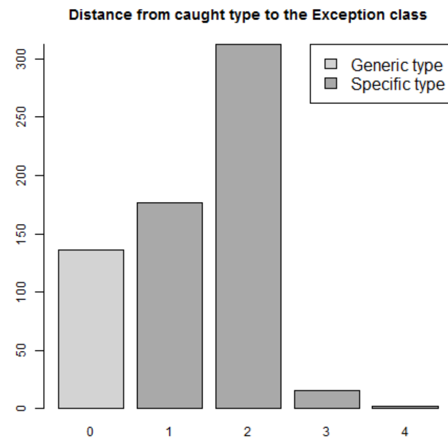



Figure A.42: Distance from the caught exceptions to the `Exception` type in Fabric

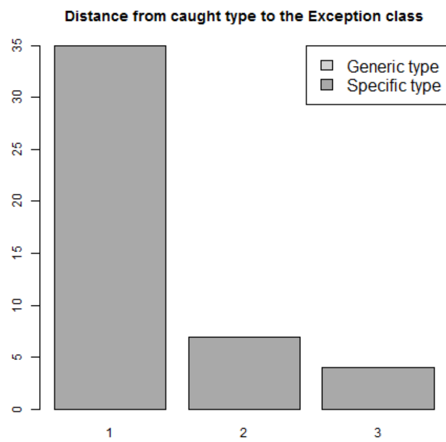



Figure A.43: Distance from the caught exceptions to the `Exception` type in Nette

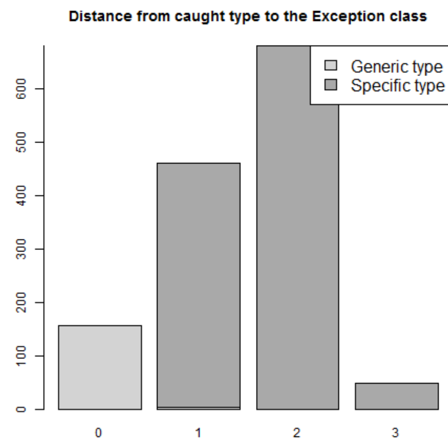



Figure A.44: Distance from the caught exceptions to the `Exception` type in Phing

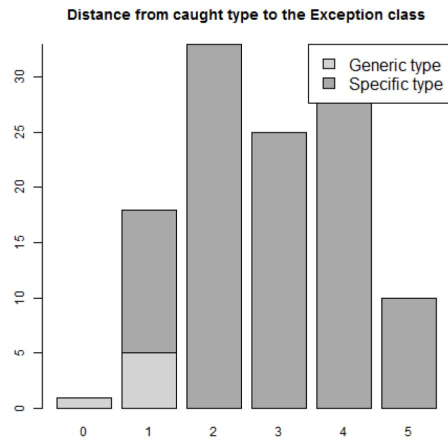



Figure A.45: Distance from the caught exceptions to the `Exception` type in phpBB

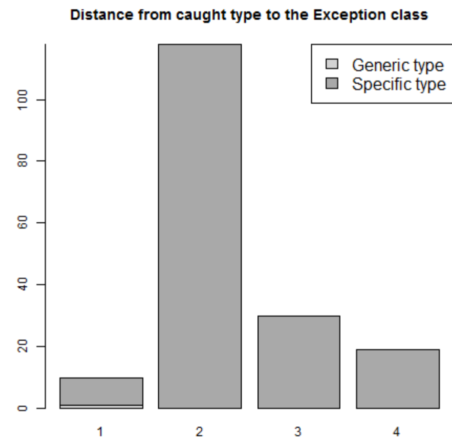



Figure A.46: Distance from the caught exceptions to the `Exception` type in phpDocumentor2

Figure A.47: Distance from the caught exceptions to the `Exception` type in PHPUnit

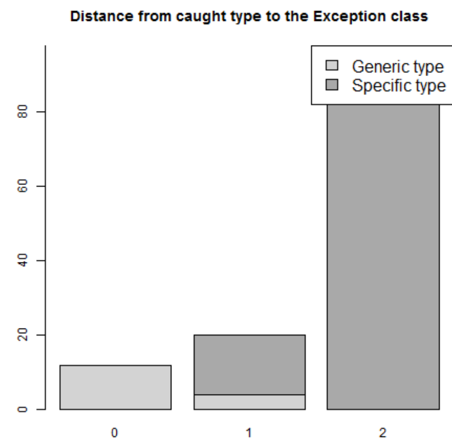



Figure A.48: Distance from the caught exceptions to the `Exception` type in Digitaalloket

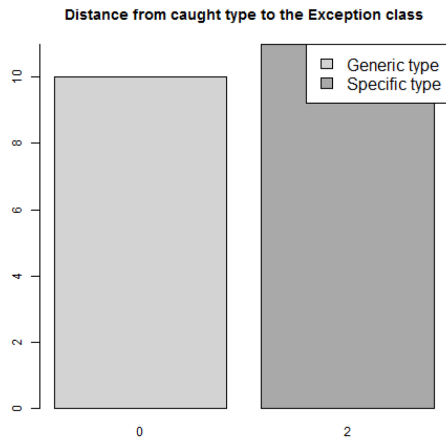



Figure A.49: Distance from the caught exceptions to the `Exception` type in Objectbrowser

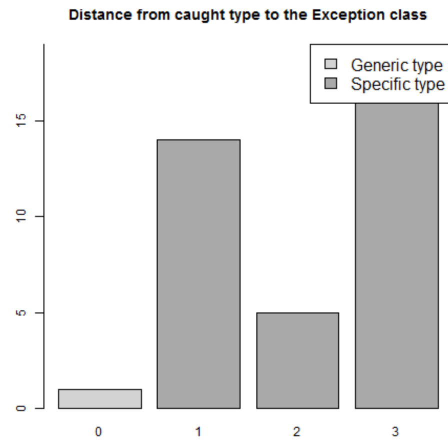



Figure A.50: Distance from the caught exceptions to the `Exception` type in Roundcube

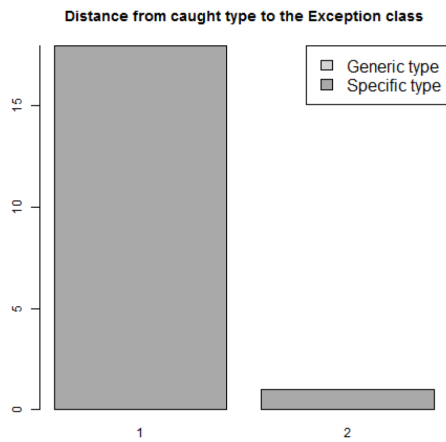



Figure A.51: Distance from the caught exceptions to the `Exception` type in Smarty

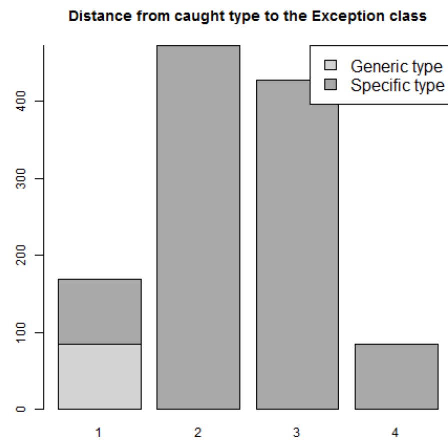



Figure A.52: Distance from the caught exceptions to the `Exception` type in Symfony

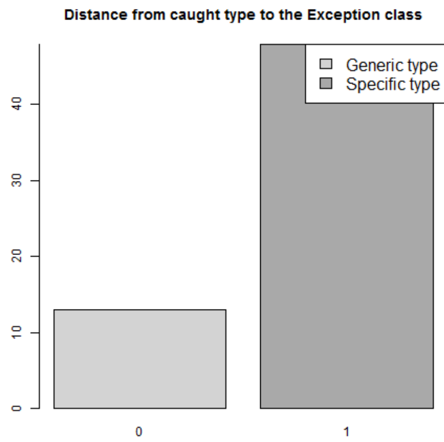



Figure A.53: Distance from the caught exceptions to the `Exception` type in Wordpress

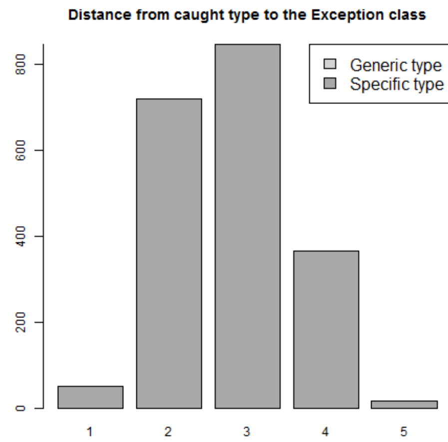



Figure A.54: Distance from the caught exceptions to the `Exception` type in Zend Framework

## A.3 Documentation Validation

Section A.3.1 shows for each project in the corpus the difference between explicitly raised exceptions and the annotated exceptions for all methods. In Section A.3.2, we show for each project the propagated and/or uncaught (but not raised) exceptions compared to the annotations for all methods. Lastly, Section A.3.3 shows for each project the annotations at contract level compared to the encountered exceptions at implementation level.

### A.3.1 Explicitly Raised Exceptions Compared To Annotations

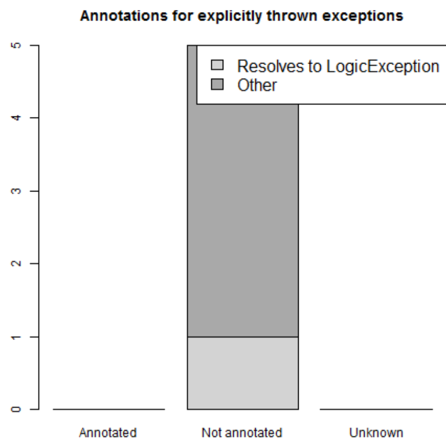



Figure A.55: Annotations compared to explicitly raised exceptions in CodeIgniter

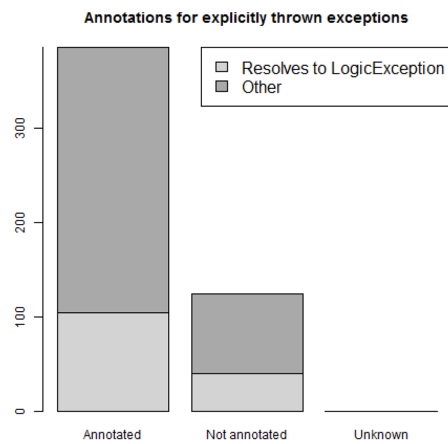



Figure A.56: Annotations compared to explicitly raised exceptions in Joomla

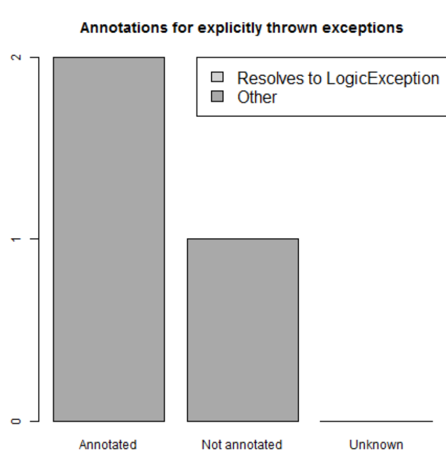



Figure A.57: Annotations compared to explicitly raised exceptions in PEAR

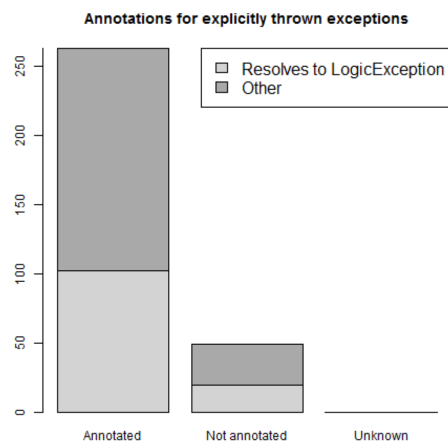



Figure A.58: Annotations compared to explicitly raised exceptions in CakePHP

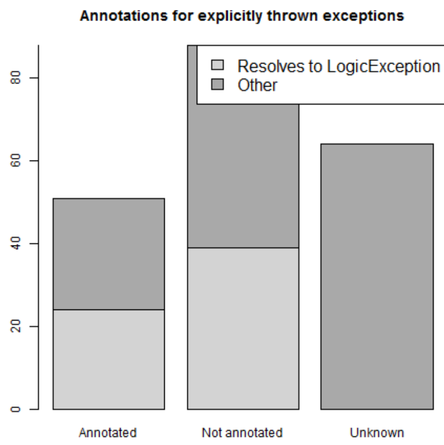



Figure A.59: Annotations compared to explicitly raised exceptions in Doctrine

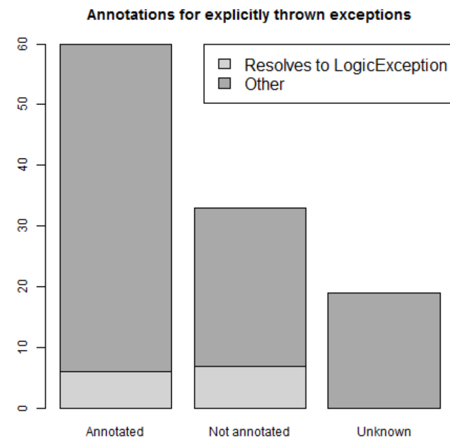



Figure A.60: Annotations compared to explicitly raised exceptions in DokuWiki

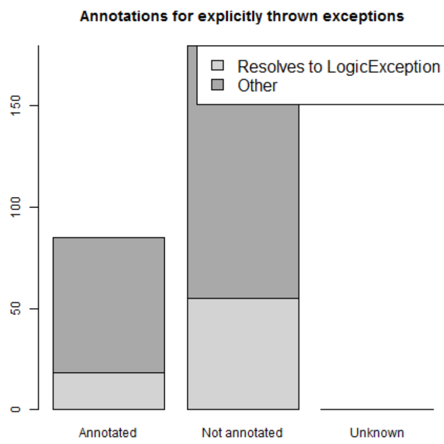



Figure A.61: Annotations compared to explicitly raised exceptions in Drupal

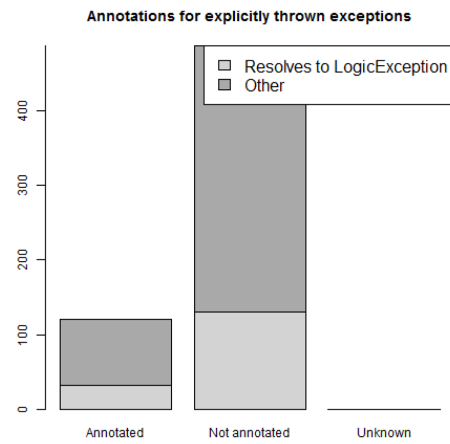



Figure A.62: Annotations compared to explicitly raised exceptions in Fabric

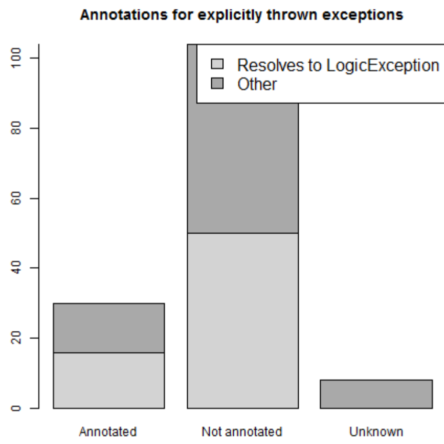



Figure A.63: Annotations compared to explicitly raised exceptions in Nette

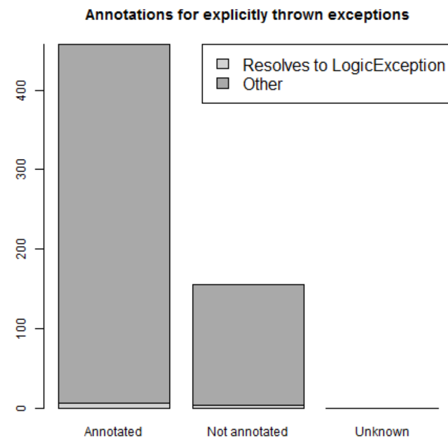



Figure A.64: Annotations compared to explicitly raised exceptions in Phing

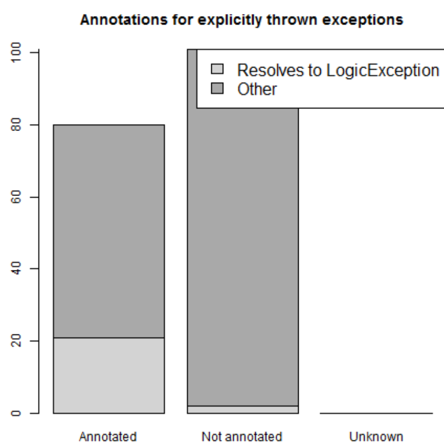



Figure A.65: Annotations compared to explicitly raised exceptions in phpBB

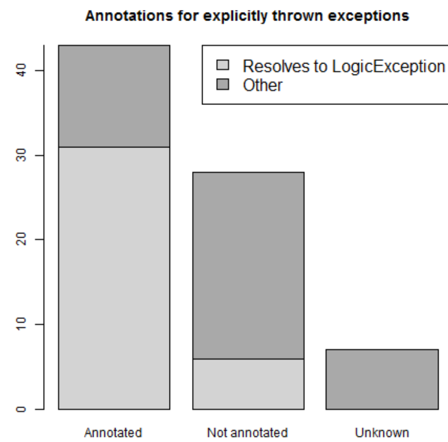



Figure A.66: Annotations compared to explicitly raised exceptions in phpDocumentor2

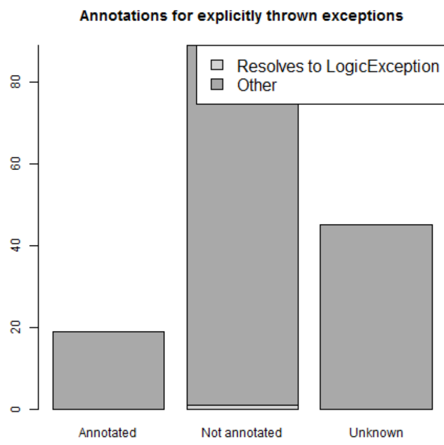



Figure A.67: Annotations compared to explicitly raised exceptions in PHPUnit

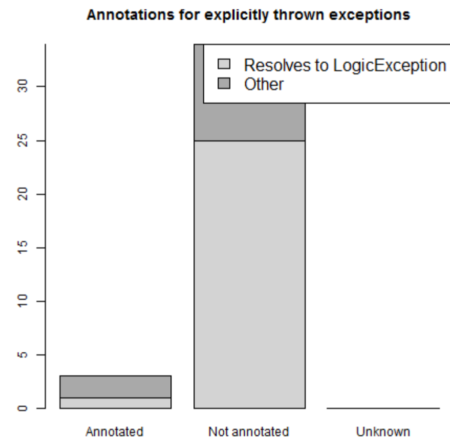



Figure A.68: Annotations compared to explicitly raised exceptions in Digitaal-loket

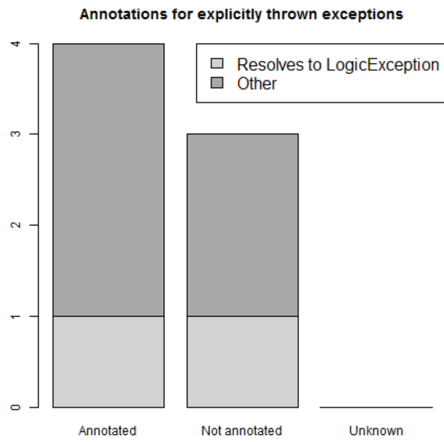



Figure A.69: Annotations compared to explicitly raised exceptions in Object-browser

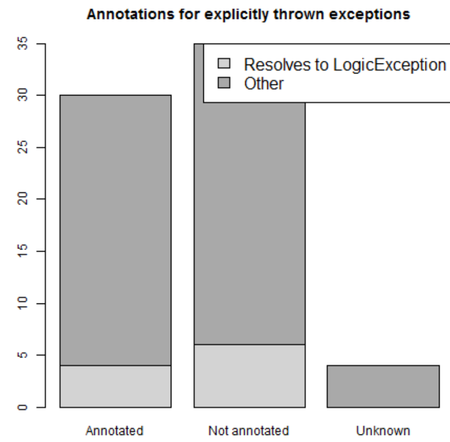



Figure A.70: Annotations compared to explicitly raised exceptions in Round-cube

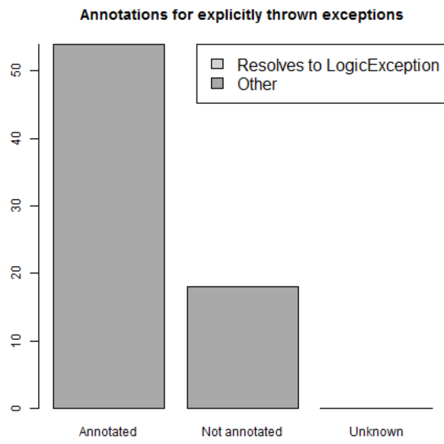



Figure A.71: Annotations compared to explicitly raised exceptions in Smarty

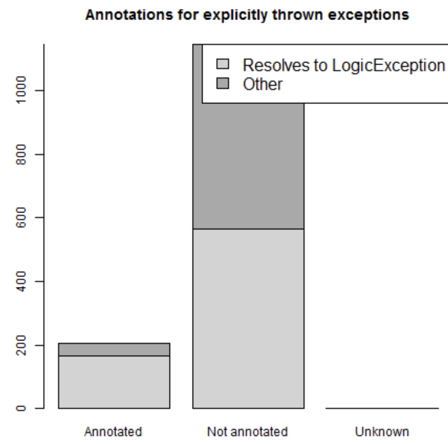



Figure A.72: Annotations compared to explicitly raised exceptions in Symfony

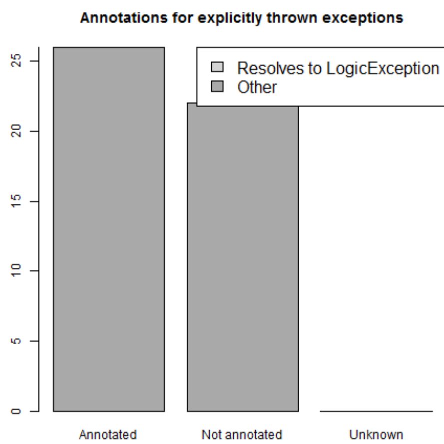



Figure A.73: Annotations compared to explicitly raised exceptions in Wordpress

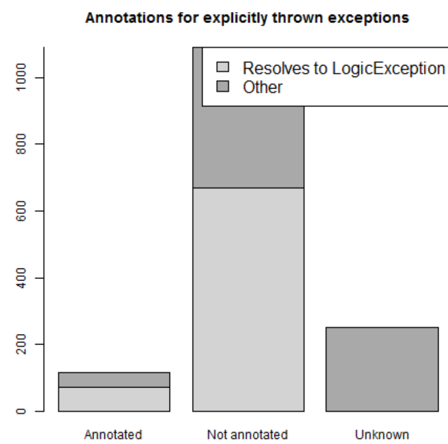



Figure A.74: Annotations compared to explicitly raised exceptions in Zend Framework

### A.3.2 Propagated or Uncaught Exceptions compared to Annotations

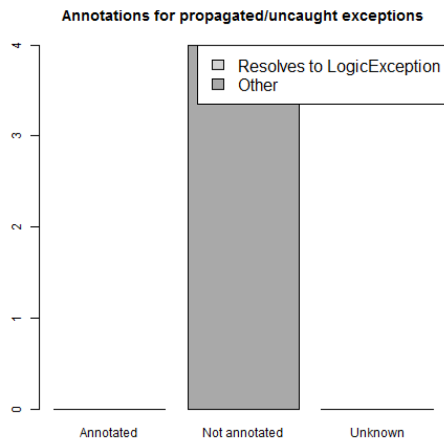



Figure A.75: Annotations compared to encountered (but not raised) exceptions in CodeIgniter

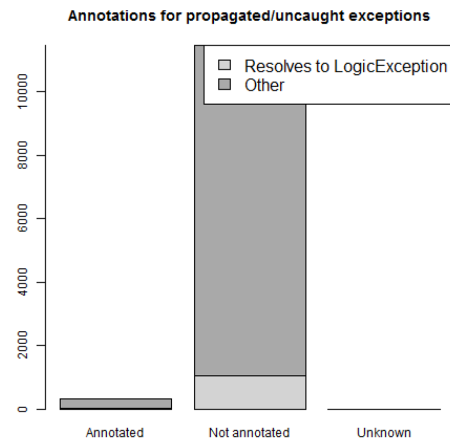



Figure A.76: Annotations compared to encountered (but not raised) exceptions in Joomla

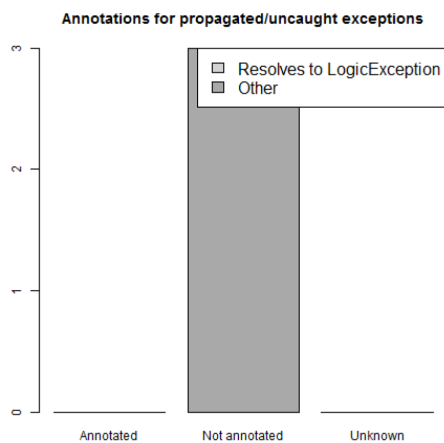



Figure A.77: Annotations compared to encountered (but not raised) exceptions in PEAR

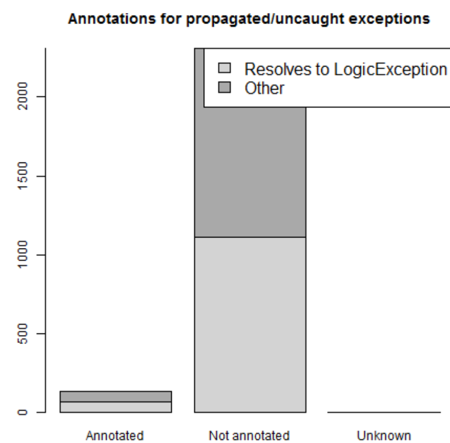



Figure A.78: Annotations compared to encountered (but not raised) exceptions in CakePHP

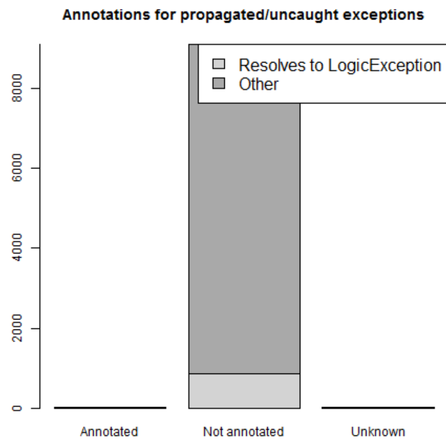



Figure A.79: Annotations compared to encountered (but not raised) exceptions in Doctrine

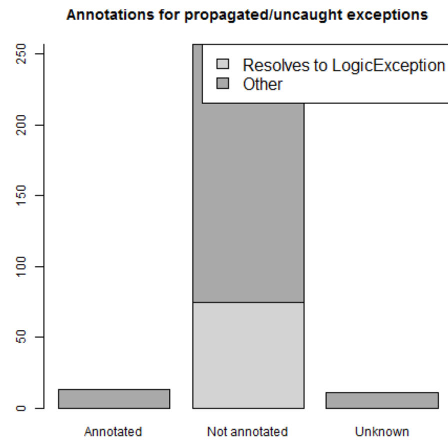



Figure A.80: Annotations compared to encountered (but not raised) exceptions in DokuWiki

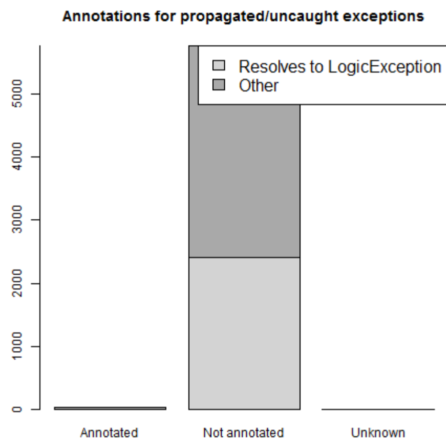



Figure A.81: Annotations compared to encountered (but not raised) exceptions in Drupal

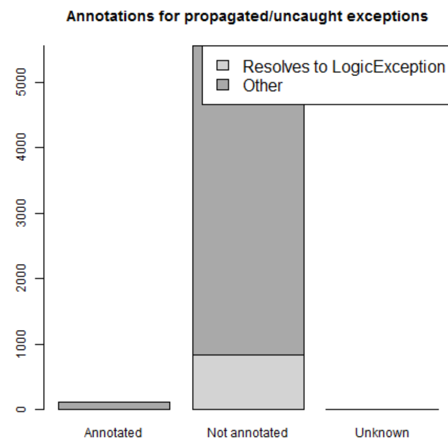



Figure A.82: Annotations compared to encountered (but not raised) exceptions in Fabric

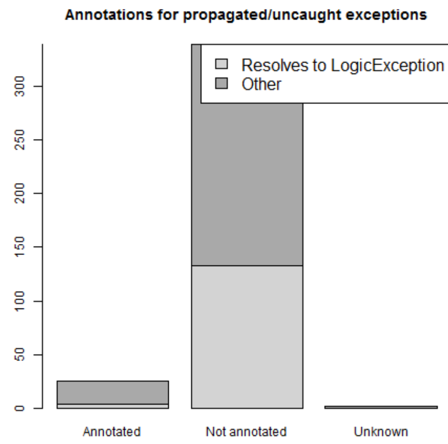



Figure A.83: Annotations compared to encountered (but not raised) exceptions in Nette

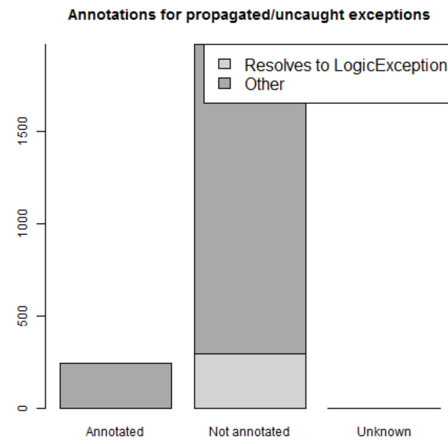



Figure A.84: Annotations compared to encountered (but not raised) exceptions in Phing

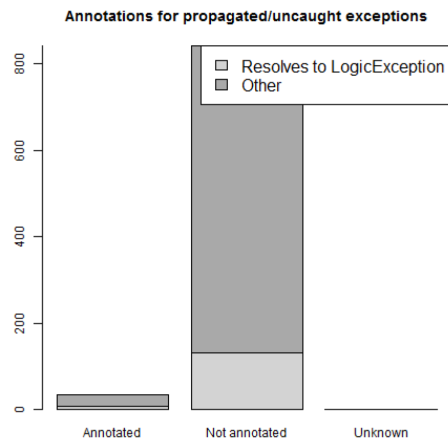



Figure A.85: Annotations compared to encountered (but not raised) exceptions in phpBB

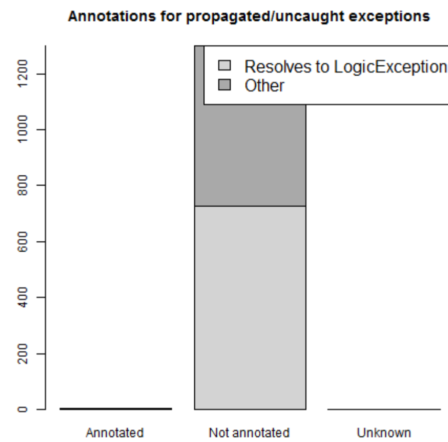



Figure A.86: Annotations compared to encountered (but not raised) exceptions in phpDocumentor2

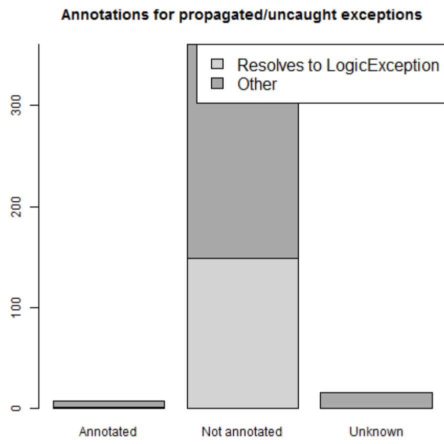



Figure A.87: Annotations compared to encountered (but not raised) exceptions in PHPUnit

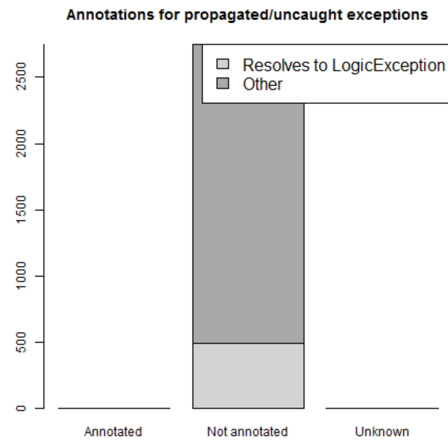



Figure A.88: Annotations compared to encountered (but not raised) exceptions in Digitaalloket

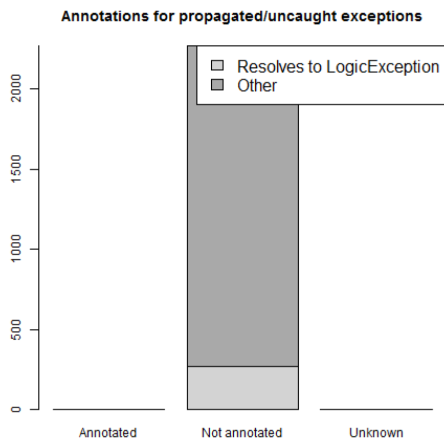



Figure A.89: Annotations compared to encountered (but not raised) exceptions in Objectbrowser

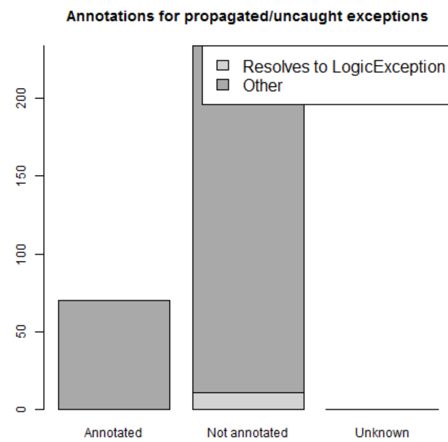



Figure A.90: Annotations compared to encountered (but not raised) exceptions in Roundcube

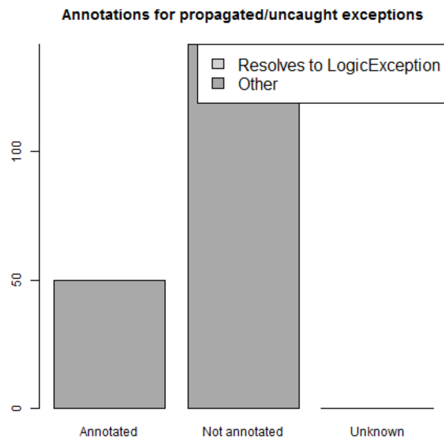



Figure A.91: Annotations compared to encountered (but not raised) exceptions in Smarty

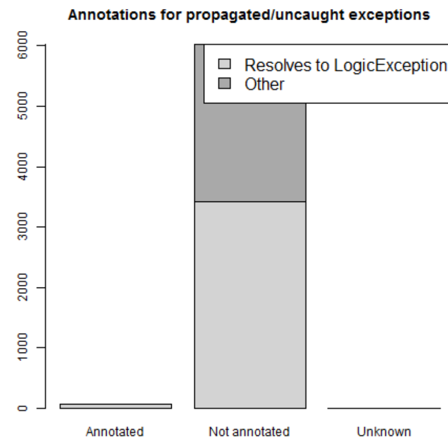



Figure A.92: Annotations compared to encountered (but not raised) exceptions in Symfony

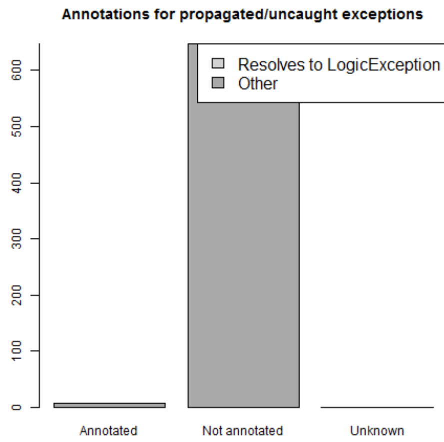



Figure A.93: Annotations compared to encountered (but not raised) exceptions in Wordpress

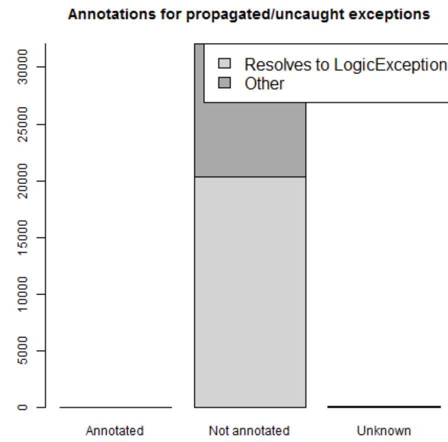



Figure A.94: Annotations compared to encountered (but not raised) exceptions in Zend Framework

### A.3.3 Annotations at Contract Level Compared to Exceptions at Implementation Level

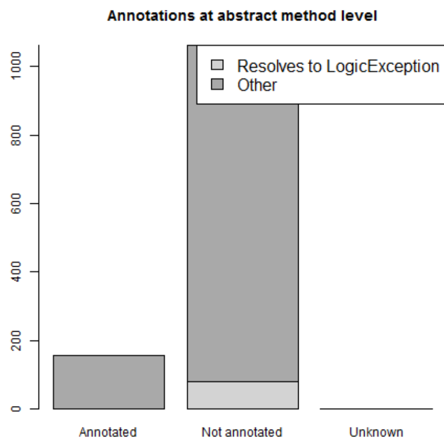



Figure A.95: Annotations at abstract method level compared to encountered exceptions in implementing methods in Joomla

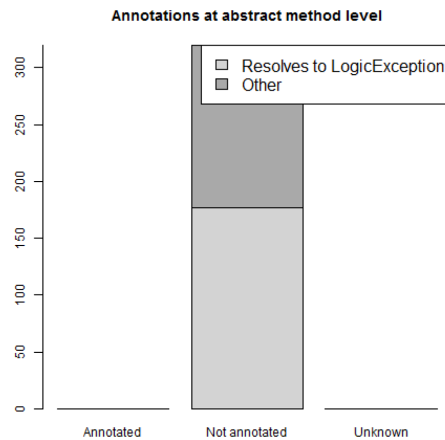



Figure A.96: Annotations at abstract method level compared to encountered exceptions in implementing methods in CakePHP

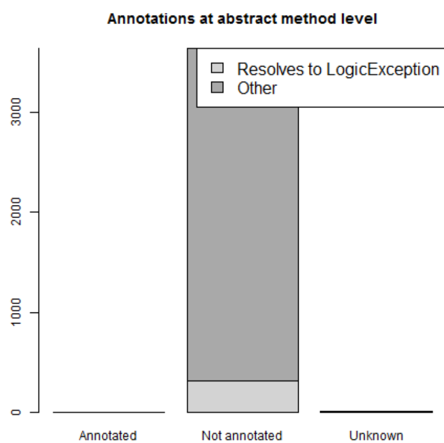



Figure A.97: Annotations at abstract method level compared to encountered exceptions in implementing methods in Doctrine

Figure A.98: Annotations at abstract method level compared to encountered exceptions in implementing methods in DokuWiki

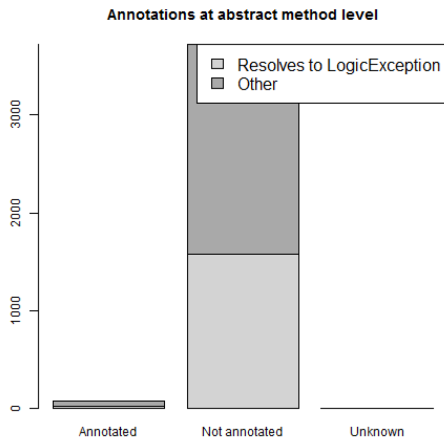



Figure A.99: Annotations at abstract method level compared to encountered exceptions in implementing methods in Drupal

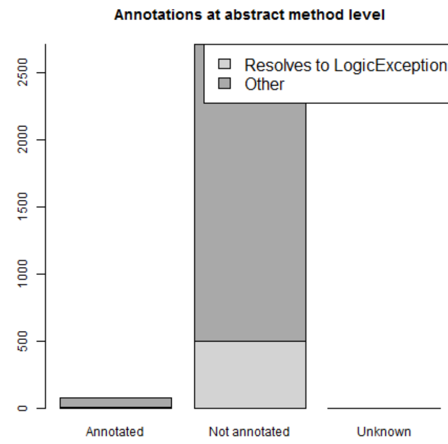



Figure A.100: Annotations at abstract method level compared to encountered exceptions in implementing methods in Fabric

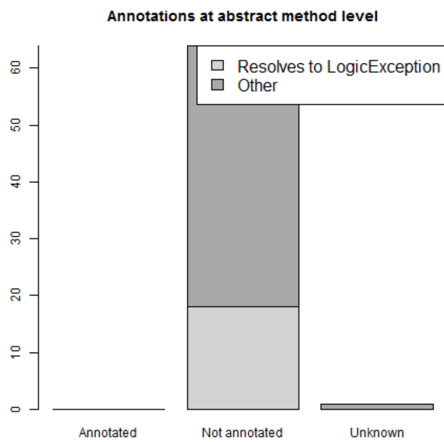



Figure A.101: Annotations at abstract method level compared to encountered exceptions in implementing methods in Nette

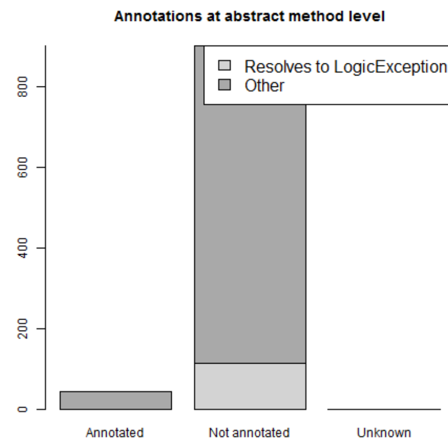



Figure A.102: Annotations at abstract method level compared to encountered exceptions in implementing methods in Phing

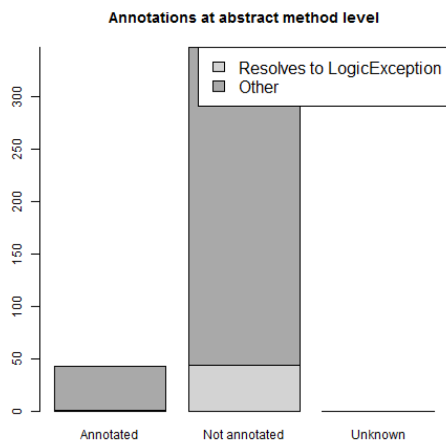



Figure A.103: Annotations at abstract method level compared to encountered exceptions in implementing methods in phpBB

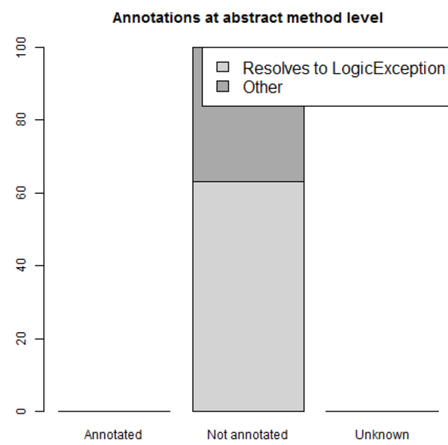



Figure A.104: Annotations at abstract method level compared to encountered exceptions in implementing methods in phpDocumentor2

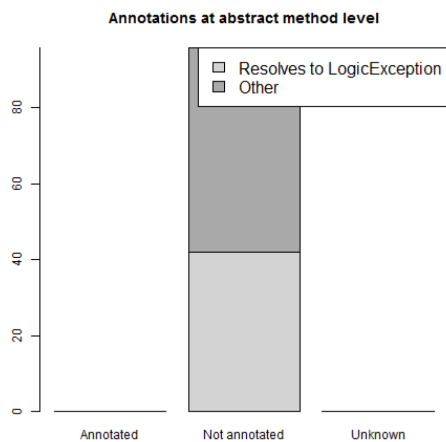



Figure A.105: Annotations at abstract method level compared to encountered exceptions in implementing methods in PHPUnit

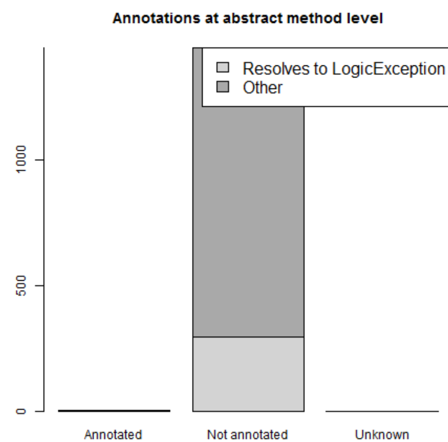



Figure A.106: Annotations at abstract method level compared to encountered exceptions in implementing methods in Digitaalloket

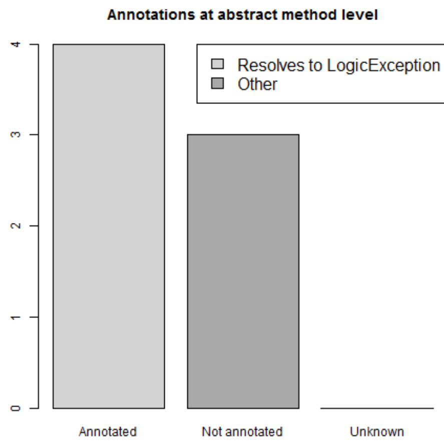



Figure A.107: Annotations at abstract method level compared to encountered exceptions in implementing methods in Objectbrowser

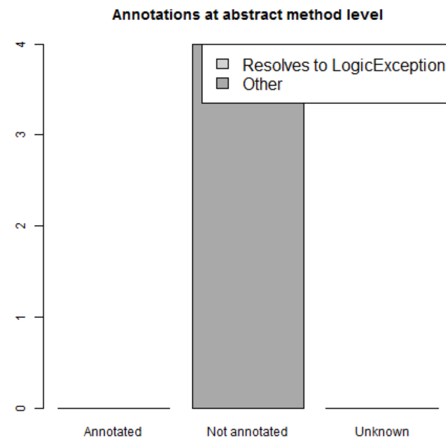



Figure A.108: Annotations at abstract method level compared to encountered exceptions in implementing methods in Roundcube

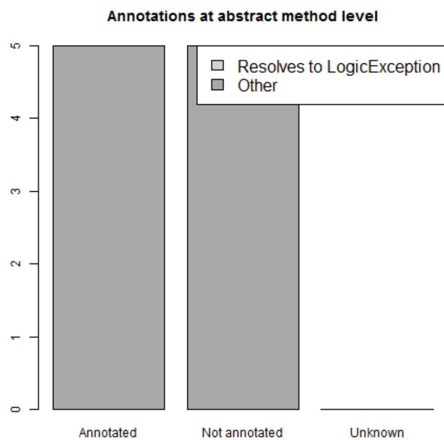



Figure A.109: Annotations at abstract method level compared to encountered exceptions in implementing methods in Smarty

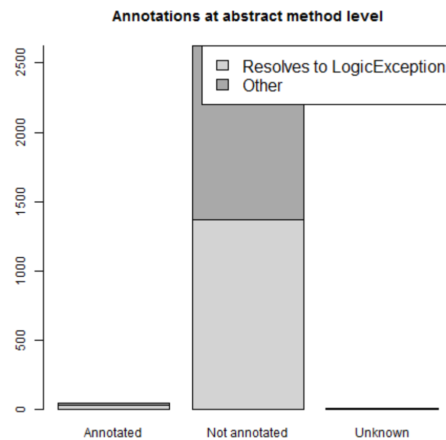



Figure A.110: Annotations at abstract method level compared to encountered exceptions in implementing methods in Symfony

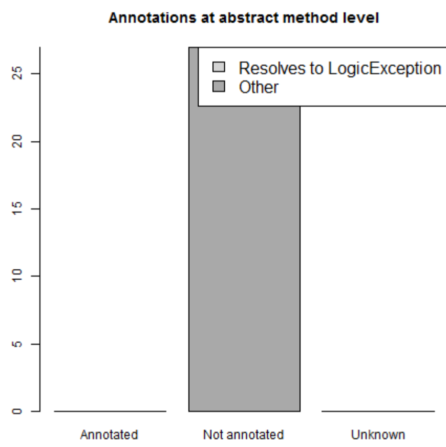



Figure A.111: Annotations at abstract method level compared to encountered exceptions in implementing methods in Wordpress

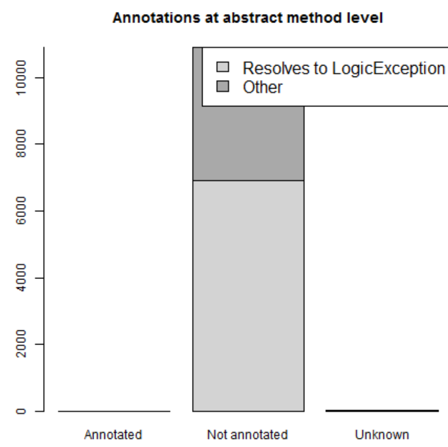



Figure A.112: Annotations at abstract method level compared to encountered exceptions in implementing methods in Zend Framework