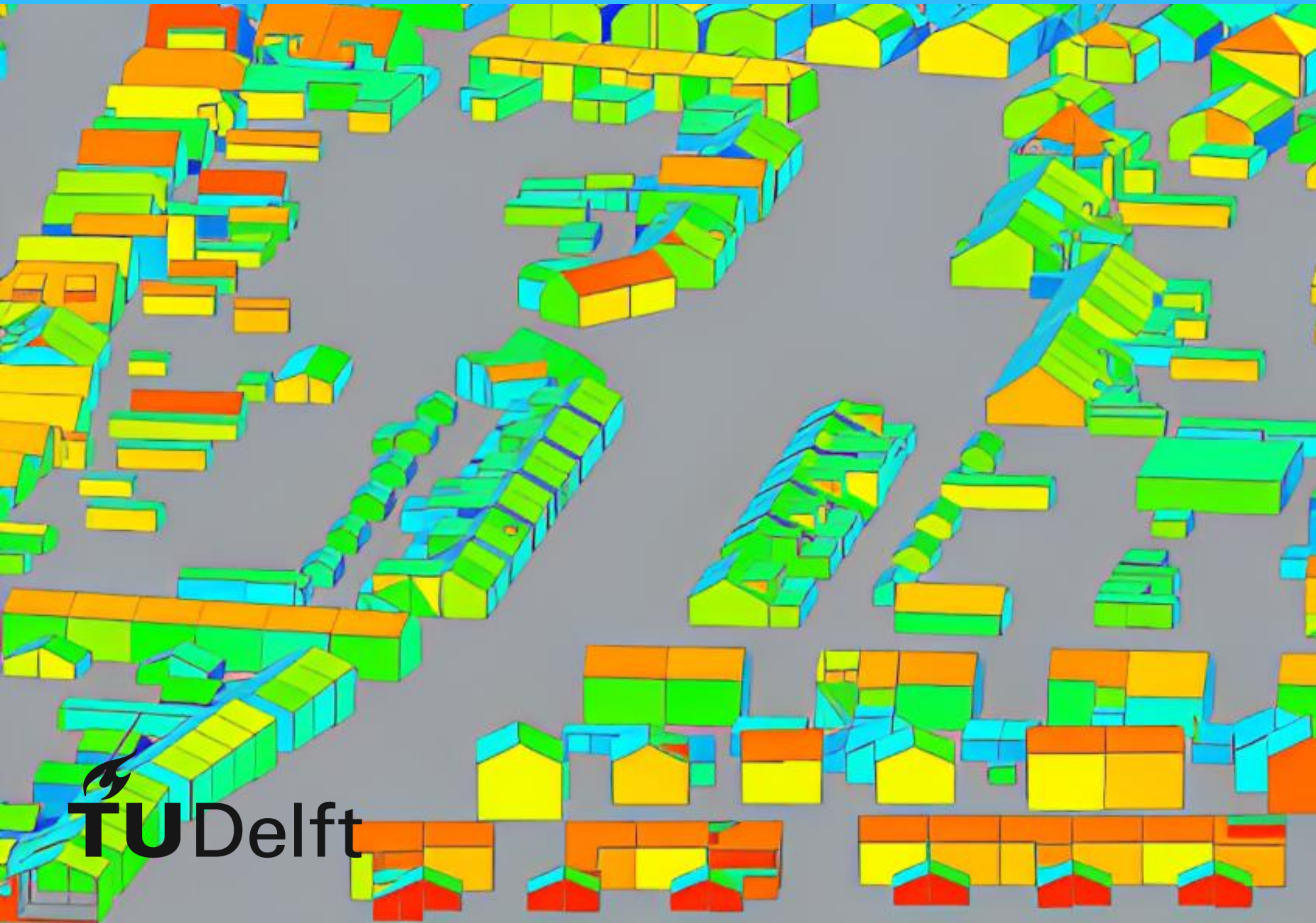
MSc thesis in Geomatics

Further Development of a QGIS Plugin for the 3D City Database

Tendai Mbwanda

2023



MSc thesis in Geomatics

Further Development of a QGIS Plugin for the CityGML 3D City Database

Tendai Mbwanda

June 2023

A thesis submitted to the Delft University of Technology in partial fulfillment of the requirements for the degree of Master of Science in Geomatics

Tendai Mbwanda: *Further Development of a QGIS Plugin for the CityGML 3D City Database* (2023)

The work in this thesis was carried out in the:

3D geoinformation group
Delft University of Technology

Supervisors:	Giorgio Agugiaro
	Camilo León Sánchez
Co-reader:	Martijn Meijers

with valuable contributions from:

Virtual City Systems
Berlin, Germany

External Supervisors:	Claus Nagel
	Zhihang Yao

Abstract

Diversity in the use cases of semantic 3D city models today is unprecedented. A key enabler for this is the City Geographic Markup Language (CityGML) standard developed by the Open Geospatial Consortium (OGC) to facilitate storing and exchanging these city models. Nevertheless, CityGML only provides object definitions which cater for a wide range of applications, making necessary the need to attach additional semantic information specific to each domain. For this reason, CityGML was designed with generic components that allow it to be extended. Alternatively, an extensibility mechanism that strengthens semantic interoperability in data exchange is the Application Domain Extension (ADE). An example is the Energy ADE which augments CityGML for Urban Energy Modelling at single-building and city-wide scales. Base CityGML datasets are commonly encoded using the Extensible Markup Language (XML), though there are other encodings based on the JavaScript Object Notation (JSON) and Structured Query Language (SQL). The latter encoding is favourable for its associated benefits that come from the underlying Relational Database Management System (RDBMS). The 3D City Database (3DCityDB), upon which this thesis is based, is one such encoding that is open source and developed for PostgreSQL and Oracle. It has a complex structure which makes it difficult for users without extensive knowledge of CityGML, databases and SQL to access data. Hence, the 3DCityDB-Tools plugin was developed to simplify user interaction with the 3DCityDB using Quantum Geographic Information System (QGIS). However, encoding an extended CityGML dataset in the 3DCityDB adds greater complexity to a system that is already complex. In addition, 3DCityDB-Tools currently has no support for Application Domain Extensions (ADEs). On this backdrop, this research was initiated to investigate the extent to which ADE support can be introduced to the 3DCityDB-Tools plugin. Its server-and-client-side components are further developed to have extended layers that interact with data in 3DCityDB tables, can be managed from the Graphical User Interface (GUI) in QGIS and whose attributes are editable. This was achieved in an incremental and iterative process while maintaining the current architecture and user experience of the plugin. Areas identified for future development relate to the underlying database encoding of CityGML and capabilities not yet supported.

Acknowledgements

I extend my gratitude to The Dutch Organisation for Internationalisation in Education (Nuffic) for their Holland Scholarship awards which sustained me for the duration of the MSc Geomatics program.

Many thanks go to my Delft University of Technology thesis supervisors, Dr. Giorgio Agugiaro and Camilo León Sánchez, for their guidance, valuable feedback and collaboration throughout this graduation project. Also, I thank the co-reader of this thesis, Martijn Meijers, for his interest in and in-depth review of my research, as well as constructive feedback. Moreover, I am grateful to my external supervisors from Virtual City Systems, Claus Nagel and Zhihang Yao. Their expertise was valuable in steering this research.

Finally, I want to express my deepest gratitude to my mother who has been incredibly supportive throughout my journey at Delft University of Technology.

Contents

List of Figures

List of Tables

Listings

List of Acronyms

3DCityDB 3D City Database

3DCM 3D city models

ADE Application Domain Extension

ADEs Application Domain Extensions

API Application Programming Interface

CityGML City Geographic Markup Language

ESRI Environmental Systems Research Institute

GIS Geographic Information System

GISs Geographic Information Systems

GUI Graphical User Interface

GUIs Graphical User Interfaces

JSON JavaScript Object Notation

LoD Level of Detail

LoDs Levels of Detail

OGC Open Geospatial Consortium

OSM OpenStreetMap

QGIS Quantum Geographic Information System

QML Qt Modelling Language

RDBMS Relational Database Management System

SFM Simple Feature Model

SQL Structured Query Language

UI User Interface

UML Unified Modelling Language

UX User Experience

XML Extensible Markup Language

1 Introduction

1.1 Motivation

Semantic 3D city models are an asset to various user groups which require them for storing and using domain-specific urban information for vast use cases (Biljecki et al., 2015). Aggregators, enablers, developers and enrichers in the geoinformation value chain (Welle Donker, 2018) often use them to exchange information and create value. To that end, CityGML was developed by the OGC to facilitate data reuse. It is an open standard which aims to establish a conventional definition of objects commonly found in the urban environment. Geometry, topology, semantics and appearance are the prominent characteristics of features in the CityGML data model, and can be represented at multiple scales and levels of detail (Gröger et al., 2012; Kolbe, 2009).

The increasing number of use cases of semantic 3D city models (3DCM) was, in addition, taken into account in the development of the standard. CityGML generalises the urban environment into several feature types shared by a wide range of applications. This allows users to attach domain-specific semantic information, as needed, to city models. To augment the semantic modelling capabilities for specific domains, the standard offers extensibility through an ADE mechanism. ADEs are data models which extend CityGML modules by defining new feature types or properties to existing ones, offering a prescribed alternative to generic city objects and attributes. Data volume increses when users enrich them with and exchange such information.

Its encapsulation of the functionality of a RDBMS makes the 3DCityDB an attractive option for encoding the data model. Yao et al. (2018) describe the mapping rules and considerations behind them, how they enable efficient management, analysis and querying of large datasets within a central data repository using the SQL as well as accessibility by external applications. One example of such a software application is the 3DCityDB-Tools, a QGIS plugin for interacting with CityGML data in a 3DCityDB instance, whose further development is investigated in this study. The 3DCityDB is similarly extensible by means of an ADE. The Energy ADE is one example which has been the subject of several studies (Agugiaro et al., 2018; Widl et al., 2021) and serves as a starting point to explore further development of 3DCityDB-Tools.

Geographic information *"... is not an end in itself but a means to support policy making as well as economic and social development ..."* (van Loenen, 2006), this includes semantic 3DCM. With increasing urbanisation globally for the last 50 years (OECD, 2020; Zhang, 2016), energy consumption and demand have soared. Buildings are reported to be responsible for at least one third of the world's energy consumption (Ahmad & Zhang, 2020; Kim et al., 2019), followed by industry and transport. "Population growth, built area increase, higher buildings services and comfort levels, together with the rise in time spent inside buildings have raised buildings consumption by 1.2%/yr since 2000." (González-Torres et al., 2022).

The Energy ADE was designed for use cases which endeavour to address these issues, providing *"a unique and standard-based data model to allow for both detailed single-building energy simulation (based on sophisticated models for building physics and occupants behaviour) and city-wide, bottom-up energy assessments, with particular focus on the buildings sector."* (Benner, 2018). Energy demand diagnostics, solar potential study and simulation of low-carbon energy strategies are cited in the Energy ADE Specification as a few use cases among many that are targeted by the extension.

Considering this context, buildings are found at the forefront of climate change mitigation and adaptation efforts, due to their high potential for improving energy efficiency and renewable energy generation (Mavromatidis et al., 2016). The European Directive 2010/31/EU on the energy performance of buildings exemplifies such efforts as it strives towards nearly net zero energy buildings. A societal drive has emerged which seeks to meet energy demand by means of research, education, finance and the law. The combination of these four streams has led to a focus on sustainable or renewable energy sources like solar. Its potential has been the subject of several studies at global (Huld et al., 2012; Korfiati et al., 2016), national (Mainzer et al., 2014), urban (Catita et al., 2014; Jakubiec & Reinhart, 2013; Martıénez-Rubio et al., 2016; Yuan et al., 2016) and (sub)district (Machete et al., 2018; Nguyen & Pearce, 2012; Redweik et al., 2013) scales.

However, the 3DCityDB has a complex structure consisting of 66 relations. Each CityGML feature has its own table, though its attributes are stored in multiple tables to respect hierarchies in the data model. Using a *CityObject* subclass such as *_AbstractBuilding* as an example, properties that are inherited from the superclass are kept in the *cityobject* table and those of *_AbstractBuilding* itself in the *building* table. A feature of the same class can also have user-defined attributes for which the *genericAttribute* class and *cityobject_genericattrib* table were designed. In addition, surface and volumetric boundary representation geometries of a feature are unnested into a single table. An important note to highlight is the deviation of the 3DCityDB encoding from the Simple Feature Model (SFM) implemented by widely used Geographic Information Systems (GISs) like QGIS and Environmental Systems Research Institute (ESRI) ArcGIS. Instead of dispersing attributes of a feature in more that one table, the SFM aims to have a single feature collection table in which each feature is represented with non-spatial attributes as well as a geometry property (Open Geospatial Consortium, 1999).

To handle this described complexity, the 3DCityDB demands Unified Modelling Language (UML) and general computer programming expertise from its users. Pantelios (2022) writes that because of this, the software may not be usable immediately and effectively. Hence, 3DCityDB-Tools for QGIS was developed to overcome this limitation for city planners and users that might not have the required technical knowledge but are accustomed to QGIS. Its open source nature and ubiquity make QGIS an attractive option for developing a user-friendly and simple GUI for handling geographic data encoded in the 3DCityDB.

Nevertheless, CityGML does not always contain all the required properties and semantics from one use case to another. This is the case for applications targeted by the Energy ADE which need to be accommodated by new classes and attributes. Transformation of ADE XML schemas into database tables, 33 for the Energy ADE KIT Profile, adds another layer of complexity to the 3DCityDB. Furthermore, 3DCityDB-Tools does not currently support ADEs, limiting the extent to which these software can be used to make the most of CityGML datasets. This research aims to overcome this limitation by exploring how 3DCityDB-Tools for QGIS can provide ADE capabilities to users while masking the underlying complexity of an extended 3DCityDB.

1.2 Research Question and Objectives

Without ADE support, 3DCityDB-Tools limits the extent to which a broader range of users can further exploit CityGML datasets. Thus, the following research question arises:

To what extent can support for Application Domain Extensions be added to

3DCityDB-Tools in its further development?

To address this question, the following objectives are relevant:

1. Conceptual definition of a strategy to add server-side support for an ADE to QGIS Package, the server-side component of 3DCityDB-Tools for QGIS.

2. Develop an ADE-enabled QGIS-Package, with focus on the Energy ADE KIT Profile.

3. Conceptual definition of a strategy to add client-side support for an ADE to the front-end of the 3DCityDB-Tools for QGIS plugin.

4. Develop an ADE-enabled 3DCityDB-Tools for QGIS front-end.

5. Contribute to further testing and extending and improving existing functionalities.

1.3 Scope of Research

This research will focus on the Energy ADE KIT Profile to extrapolate how 3DCityDB-Tools can support any other ADE developed for specific applications and not generic ADEs that supplement CityGML without a specific intended application. It will also be limited to versions 2.0 of the CityGML standard and 4.4 of the 3D City Database. The entire pipeline is examined in this study, from server-side constructs which leverage database management system functionalities in handling ADE information to feature retrieval and editing on the client-side.

1.4 Research Organisation

The current chapter presents a background to the research conducted. Chapter 2 analyses concepts and tools relevant to this research based on literature. Following this, the methodology that guided further development of 3DCityDB-Tools is discussed in chapter 3. An account of how ADE support was explored in the plugin is given in chapter 4. Lastly, chapter 5 sums up the research and points out a few recommendations for future work.

2 Theoretical Background and Related Work

2.1 Theoretical Background

2.1.1 CityGML

CityGML is an open standard developed by the OGC to establish a conventional definition of topographic objects commonly found in the urban environment. Its data model groups them into packages "according to thematic and logical criteria and not according to graphical or rendering considerations" (Yao et al., 2018). Figure 1 shows the thematic decomposition of CityGML version 2.0 into the core module and thematic extension modules. Within CityGML core, "CityGML uses a subset of the GML3 geometry model which is an implementation of the ISO 19107 standard" (Kolbe, 2009), allowing objects to be represented by aggregate or composite 0D, 1D, 2D or 3D geometric primitives. CompositeSurface and MultiSurface are respective examples of a composite and an aggregate geometry. The thematic modules are defined by this geometry and topology, as well as semantics and properties.

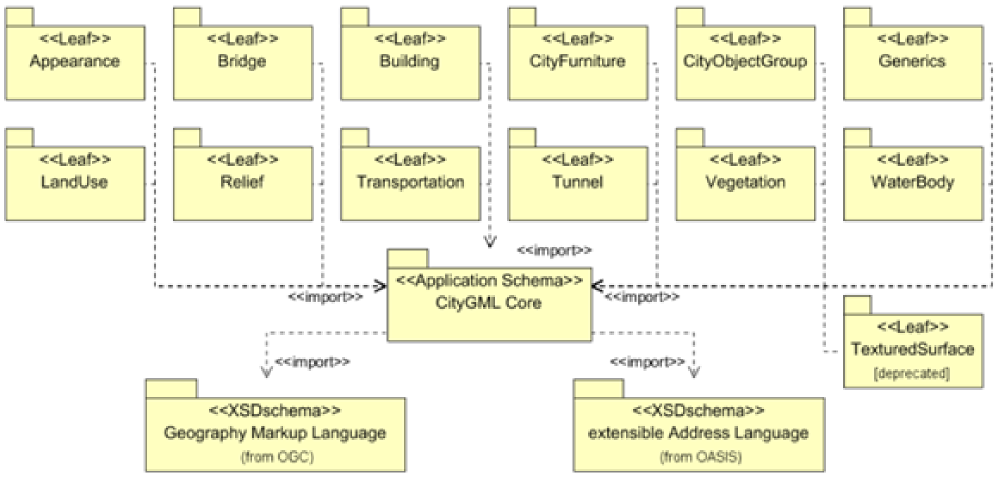


Figure 1: *"UML package diagram illustrating the separate modules of CityGML and their schema dependencies. Each extension module (indicated by the leaf packages) further imports the GML 3.1.1 schema definition in order to represent spatial properties of its thematic classes."* Source: (Gröger et al., 2012)

The Building module depicted in Figure 2 can be used as an illustration. A building can have at least one aggregate or composite geometry at

Levels of Detail (LoDs) between 0 and 4 for the former, and between 1 and 4 for the latter. Composite geometry objects imply the preservation of topology, while aggregate geometry objects are not restricted in terms of spatial relationships. "At the semantic level, real-world entities are represented by features, such as buildings, walls, windows, or rooms." (Gröger et al., 2012). Attributes are housed by classes of these real world entities as Figure 2 demonstrates with *_AbstractBuilding*. Conventional definition of topographic objects drills down to values of some of their attributes. CityGML achieves this using enumerations and codelists. Enumerations are immutable and standardised values of a particular property of a feature. For instance, *RelativeToTerrainType* (Figure 2) which prescribes a list of values for the *relativeToTerrain* property of a *CityObject*. Codelists resemble enumerations, except they also allow arbitrarily specified attribute values. Though CityGML does not specify any, its extensions may offer value lists which the user can add to and the Energy ADE is one example.

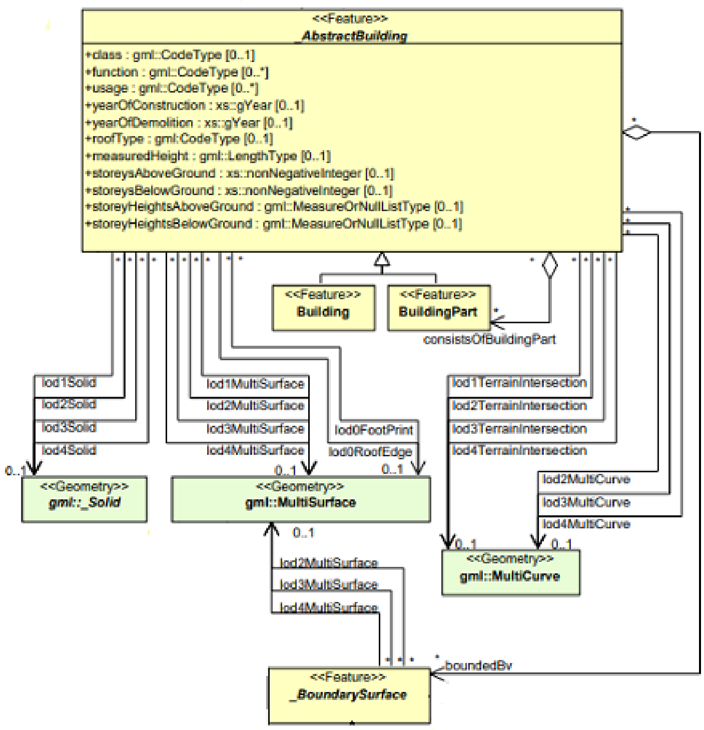
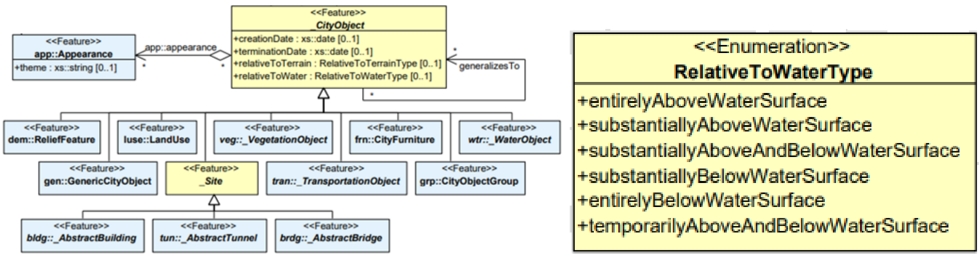


Figure 2: UML diagram of the Building module along with its associated geometry. (Figure adapted from (Gröger et al., 2012))



(a) The *CityObject* class and its subclasses that inherit its properties.

(b) Enumeration values for the *relativeToWater* property of a *CityObject*.

Figure 3: *CityObject* subclasses which inherit the *relativeToWater* property, and the *relativeToWaterType* values specified by CityGML.

CityGML can be extended by creating new object types or adding new properties. This is made possible by the *_genericAttribute* data type and *GenericCityObject* class contained in the *Generics* module (Figure 4). However, these mechanisms limit interoperability as additional name-value pair properties and semantics cannot be stored in a systematic way. An alternative way to extend the data model is through ADEs. (Biljecki et al., 2018) distinguish between two types of ADEs, those developed to support a specific application and generic ADEs.

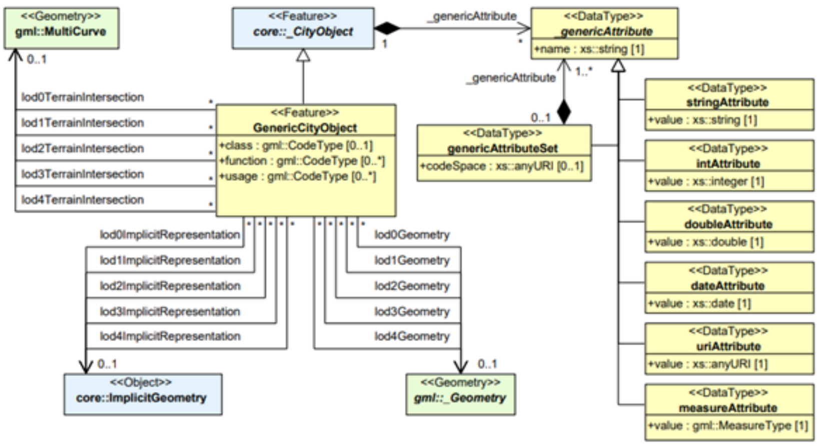


Figure 4: *" UML diagram of generic objects and attributes in CityGML. Prefixes are used to indicate XML namespaces associated with model elements. Element names without a prefix are defined within the CityGML Generics module."* Source: (Gröger et al., 2012)

2.1.2 Energy ADE

The Energy ADE specialises CityGML for a variety of Energy applications (Benner, 2018). It provides a formalised data model, currently at version 1.0, which fosters interoperability and allows for ”both detailed single-building energy simulations and city-wide bottom-up energy assessment” (Agugiaro et al., 2018). One of the two objectives behind its design is as the previously quoted study states to ”provide data to assess the energy performance of buildings ...”. To further develop 3DCityDB-Tools for QGIS, this provided an operating scope with respect to CityGML.

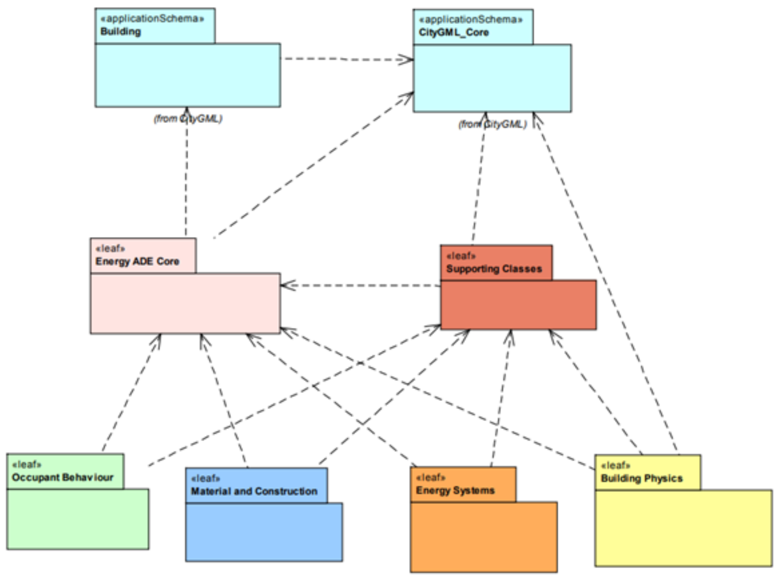
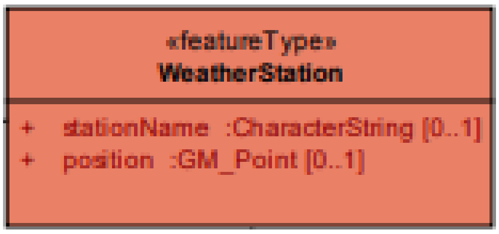


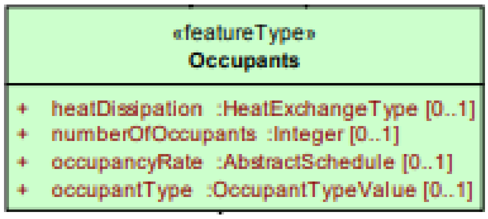
Figure 5: Package diagram of the Energy ADE data model. Source: (Benner, 2018)

Similarly, the Energy ADE has a core module, as well as four other thematic modules containing energy-related entities and attributes. Another package designated as Supporting Classes defines types which allow representation of temporal attributes. Figure 5 depicts a package diagram of the Energy ADE. It augments CityGML with new attributes and classes which may be *CityObjects*. *_AbstractBuilding* with the stereotype *ADEElement* (Figure 7a) is an example of the former, *WeatherStation* and *Occupants* are examples of the latter. *_AbstractBuilding* provides additional properties to the native CityGML class of the same name.

WeatherStation and *Occupants* are new classes altogether, though one is a *CityObject* (Figure 6a) and the other is not (Figure 6b). All classes that are not *CityObject* descendants were put in place to capture supplementary information about static attributes and temporal values of other classes that may be. *HeightAboveGround* which provides *_AbstractBuilding* with *heightReference* and *value* (Geiger et al., 2018), as well as *VolumeType* which supports *ThermalZone* with *type* and *value* are examples of classes that enrich *CityObjects*. *OpticalProperties*, a *non-CityObject*, harvests *fraction* and *waveLengthRange* from *Transmittance*. These three classes were used to demostrate for their similarity to *_genericAttribute* type in the CityGML data model, which is already supported by 3DCityDB-Tools. To complement all these Energy ADE components, the data model also specifies its own codelists and enumerations.



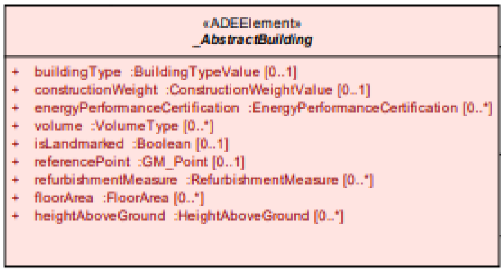
(a) WeatherStation, a new *CityObject* introduced by the Energy ADE.



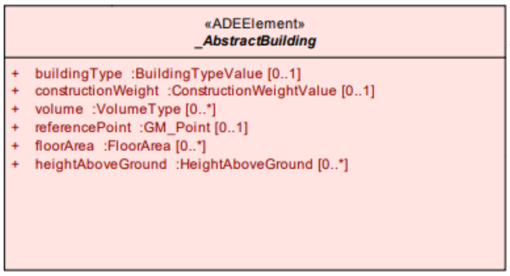
(b) Occupants, a new *non-CityObject* introduced by the Energy ADE.

Figure 6: Two new classes in the Energy ADE that extend CityGML. Source: (Benner, 2018)

Moreover, some information in the Energy ADE is not required for every application in the Energy domain. Instead, only a subset of what the ADE offers may be used in some use cases, giving rise to the specialisation of the ADE itself. The Energy ADE KIT Profile is one such product which removes unwanted classes, attributes, enumerations and codelists and leaves only those that are necessary for a given use case. To illustrate, reference can be made to Figure 7. Figure 7a shows the *_AbstractBuilding* class in the full Energy ADE, whereas for the same class shown in Figure 7b some attributes are left out of the KIT Profile. For clarification, this research focuses on the Energy ADE KIT Profile, or simply the KIT Profile, though reference may be made to the whole Energy ADE.



(a) *_AbstractBuilding* class in the full Energy ADE. Source: (Benner, 2018)



(b) *_AbstractBuilding* class in the KIT Profile.

Figure 7: *_AbstractBuilding* in the full Energy ADE and KIT Profile.

2.2 Related Work

2.2.1 The 3D City Database

The "3DCityDB is an Open Source software suite allowing to import, manage, analyze, visualize, and export virtual 3D city models according to the CityGML standard, supporting both versions 2.0 and 1.0." (Yao et al., 2018). 3DCityDB 4.4.0, the latest release at the time of writing, maps classes in CityGML to 66 database relations in a PostgreSQL database schema. This schema also brings various functionalities for computations on geometries, spatial indexing, deleting *CityObject* instances from the database and other trivial data management tasks (The 3D City Database, n.d.). Aside from the database itself, another component of the software suite is the 3D City Database Importer/Exporter, a tool that implements the above mapping. One of its functions is to ensure that a file-encoded CityGML dataset is correctly imported into a 3DCityDB schema.

Stadler et al. (2009) highlight the differences in relations and hierarchies from one CityGML module to another, making necessary the development of some criteria under which a relational model is created. Four mapping guidelines are presented by Yao et al. (2018) which optimise operating performance and semantic interoperability.

CityGML inheritance hierarchies are placed in the same table to enable quick retrieval of *CityObjects*. Classes at the same inheritance level are mapped to the table of their common superclass, having *Building* and *BuildingPart* in one relation for example, to boost overall performance by avoiding joining queries. However, this rule is only applied when the superclass is abstract and holds all attributes and relationships inherited by its subclasses which in turn do not have any additional properties or associations. Constraining the mapping approach this way is an attempt to maximise storage efficiency in case there are subclasses that have very different attributes and cause a large number of empty cells in the table. Aggregations and compositions such as those between *_BoundarySurface* features and geometry types such as *MultiSurface* are also unnested into one relation, linked by the columns `parent_id` and `root_id` in this example. Doing so works around recursive joins which reduce database performance. The previous example can be extended to the last rule, mapping of boundary representation geometries onto a single table. Overall, these four guidelines flatten the complex CityGML data model, an approach also highlighted and undertaken by Ledoux et al. (2019) in the design of a JSON-based encoding of the same standard.

CityGML extensibility mechanisms are also supported by the 3DCityDB. Classes in its *Generics* module (Figure 4) which allow creating new *City-Objects* or enriching existing ones with new attributes compose 2 of the 66 tables. Formalised extensions are mapped into multiple relations, for instance 33 more are introduced by the Energy ADE KIT Profile, using the ADE Manager in the 3D City Database Importer/Exporter which is shown in Figure 8. A database prefix is required for each ADE. One use for the prefix is naming tables such that the table for the *ThermalZone* class is named as *ng_thermalzone*. In Figure 8, the prefix *ng* is input for the Energy ADE for naming, and will be used throughout this report for the KIT Profile. After extending the 3DCityDB, an extended dataset can then be imported, though not using the same four rules mentioned above for this particular ADE. Not all abstract classes have attributes, some classes at the same inheritance level have different properties (Figure 9), plus aggregations and compositions may not be flattened into the same table as is the case for the *ThermalZone* and *ThermalBoundary* classes (Figure 10).

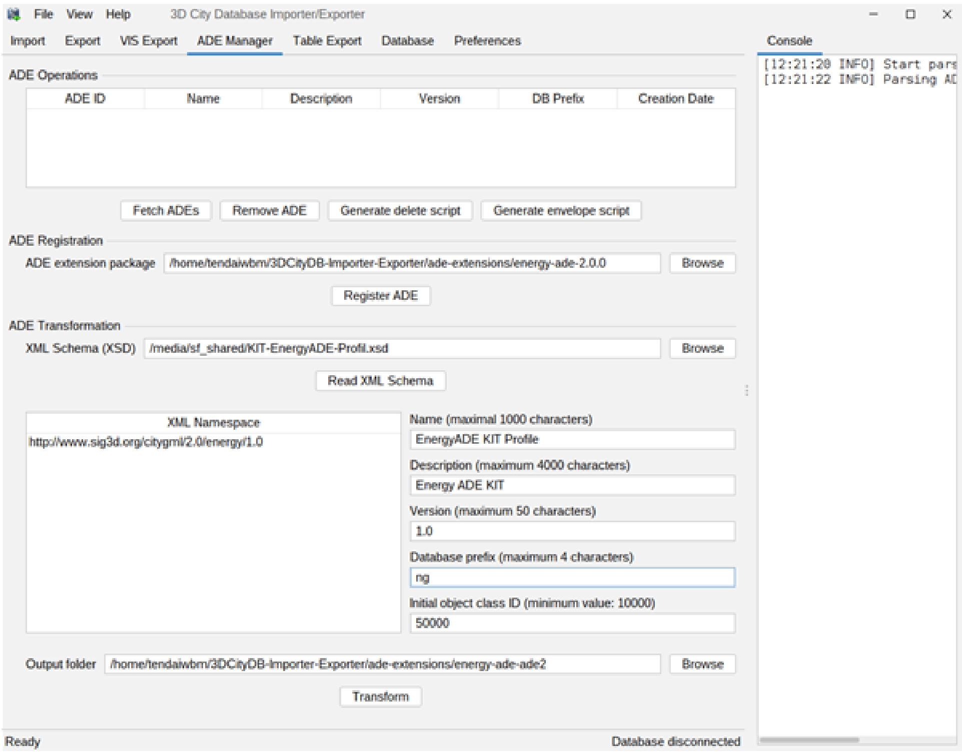


Figure 8: ADE Manager in the 3D City Database Importer/Exporter.

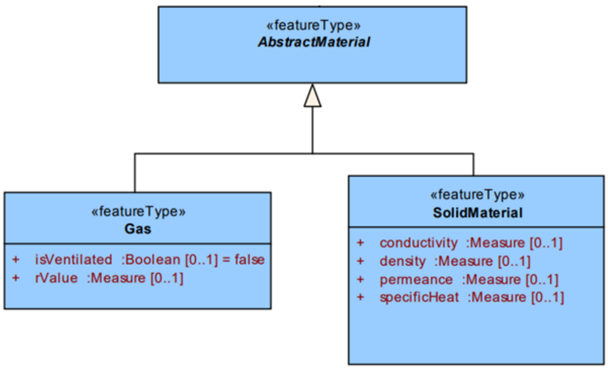


Figure 9: Extract of EnergyADE KIT Profile showing *AbstractMaterial* and its subclasses *Gas* and *SolidMaterial* at the same inheritance level.

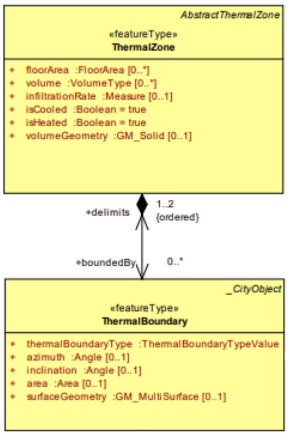


Figure 10: A composition relationship between *ThermalZone* and *Thermal-Boundary* defined by the Energy ADE KIT Profile.

2.2.2 QGIS

QGIS is an open source Geographic Information System (GIS) which allows creating, querying, visualisation and processing of geographic data (Sherman et al., 2005). In this regard, it interacts with a plethora of vector and raster data formats. "While its initial goal was simply to develop a spatial data viewer, it has now evolved into a complete platform for the loading, transformation, and processing of spatial data..." (Vitalis et al., 2020). The Qt framework, Python and C++ are used to build QGIS. With this comes the possibility to extend its functionality, taking advantage of its Application Programming Interface (API) interface for either of the aforementioned programming languages. These customised functionalities are termed QGIS plug-ins and 3DCityDB-Tools is one example among many listed in the QGIS Python Plugins Repository (QGIS, 2023).

Due to a greater complexity brought to the 3DCityDB by the Energy ADE KIT Profile, this research strives to produce ADE feature collection tables which adhere to the SFM. Through 3DCityDB-Tools, these tables can then be easily imported and managed in QGIS using the *QgsVectorLayer* and *QgsDataSourceUri* classes in its API. A *QgsVectorLayer* instance creates *"... a vector layer which manages a vector based data sets"* (QGIS-Python-API, 2018). The QGIS Python API documentation also describes that a *QgsVectorLayer* object is associated to a *QgsDataSourceUri* which can connect to a PostgreSQL data source using the connection parameters, table, geometry column, and other attributes. This implies that the data source must have its spatial and non-spatial properties in the same table. Having ADE features in the 3DCityDB organised in this way would enable them to be imported into QGIS as instances of the *QgsVectorLayer* class.

Datasets imported into QGIS are simply referred to as *Layers* and are placed in the *Layers* panel. Additional organisation can be performed such that layers are structured in a tree, whose nodes are based on a user-defined criteria. Figure 11 illustrates a layer tree which organises layers with a PostgreSQL data source based on a database name, schema name, as well as CityGML feature type and level of detail. Each layer has an *Attribute Table* which is a collection of feature attributes. Its rows represent features in the dataset, and columns contain feature attribute values as shown in Figure 12. Feature editing can be performed directly from the *Attribute Table*. New features can be created, and existing ones can be deleted or their attributes modified. Any updates made on a layer in QGIS are committed to the data source as well.

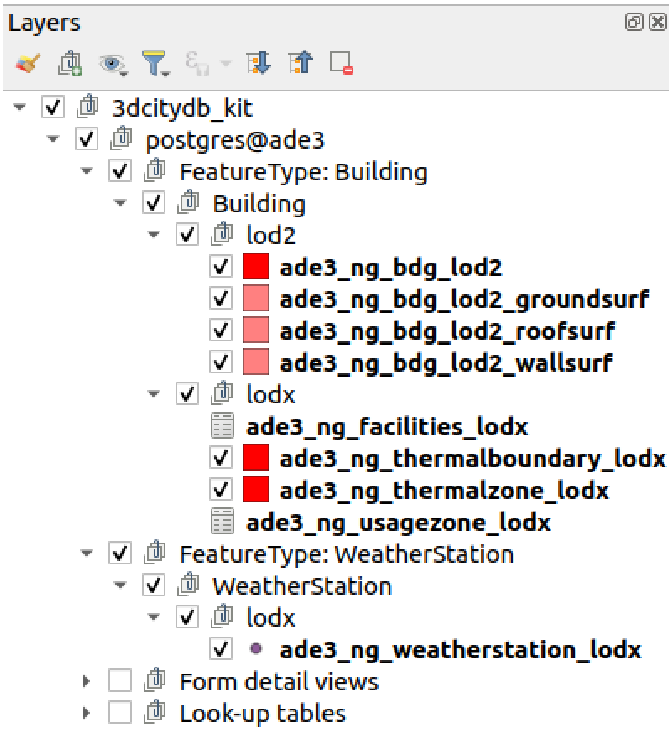
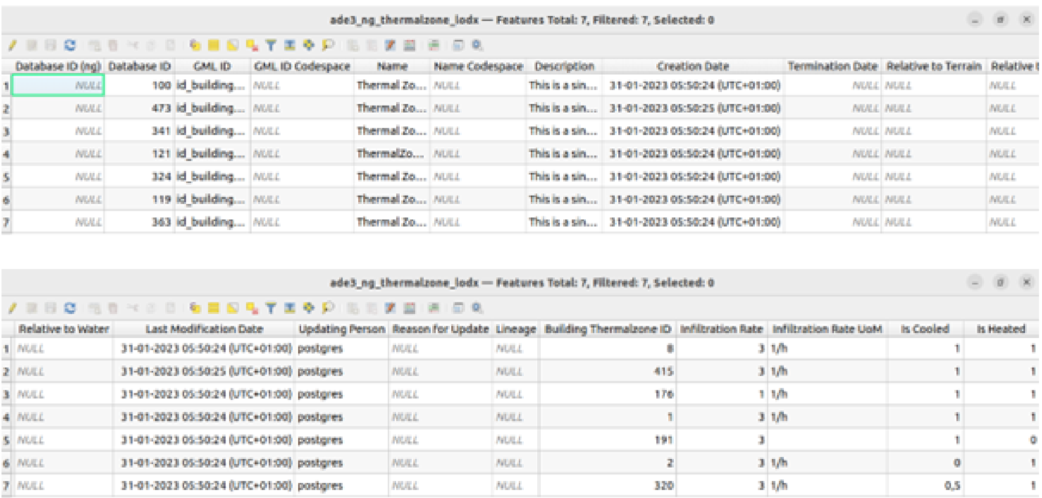


Figure 11: QGIS Layers panel with layers structured in a tree.



ade3_ng_thermalzone_lodx — Features Total: 7, Filtered: 7, Selected: 0

	Database ID (ng)	Database ID	GML ID	GML ID Codespace	Name	Name Codespace	Description	Creation Date	Termination Date	Relative to Terrain	Relative
1	NULL	100	id_building...	NULL	Thermal Zo...	NULL	This is a sin...	31-01-2023 05:50:24 (UTC+01:00)	NULL	NULL	NULL
2	NULL	473	id_building...	NULL	Thermal Zo...	NULL	This is a sin...	31-01-2023 05:50:25 (UTC+01:00)	NULL	NULL	NULL
3	NULL	341	id_building...	NULL	Thermal Zo...	NULL	This is a sin...	31-01-2023 05:50:24 (UTC+01:00)	NULL	NULL	NULL
4	NULL	121	id_building...	NULL	ThermalZo...	NULL	This is a sin...	31-01-2023 05:50:24 (UTC+01:00)	NULL	NULL	NULL
5	NULL	324	id_building...	NULL	Thermal Zo...	NULL	This is a sin...	31-01-2023 05:50:24 (UTC+01:00)	NULL	NULL	NULL
6	NULL	119	id_building...	NULL	Thermal Zo...	NULL	This is a sin...	31-01-2023 05:50:24 (UTC+01:00)	NULL	NULL	NULL
7	NULL	363	id_building...	NULL	Thermal Zo...	NULL	This is a sin...	31-01-2023 05:50:24 (UTC+01:00)	NULL	NULL	NULL

ade3_ng_thermalzone_lodx — Features Total: 7, Filtered: 7, Selected: 0

	Relative to Water	Last Modification Date	Updating Person	Reason for Update	Lineage	Building Thermalzone ID	Infiltration Rate	Infiltration Rate UoM	Is Cooled	Is Heated
1	NULL	31-01-2023 05:50:24 (UTC+01:00)	postgres	NULL	NULL	8	3	1/h	1	1
2	NULL	31-01-2023 05:50:25 (UTC+01:00)	postgres	NULL	NULL	415	3	1/h	1	1
3	NULL	31-01-2023 05:50:24 (UTC+01:00)	postgres	NULL	NULL	176	1	1/h	1	1
4	NULL	31-01-2023 05:50:24 (UTC+01:00)	postgres	NULL	NULL	1	3	1/h	1	1
5	NULL	31-01-2023 05:50:24 (UTC+01:00)	postgres	NULL	NULL	191	3		1	0
6	NULL	31-01-2023 05:50:24 (UTC+01:00)	postgres	NULL	NULL	2	3	1/h	0	1
7	NULL	31-01-2023 05:50:24 (UTC+01:00)	postgres	NULL	NULL	320	3	1/h	0,5	1

Figure 12: Attribute Table of a *QgsVectorLayer* in QGIS.

A *QgsVectorLayer* has several other properties which include *Symbology* and an *Attributes Form*. *Symbology* defines how features are displayed in the QGIS map canvas and *Layers* panel. An *Attributes Form* is similar to the *Attribute Table*, except that it lays out attributes in a view that is more user friendly. When a layer is loaded in QGIS, a default *Attributes Form* is created using *"...a user interface specification and programming language"* (Qt-Project, n.d.) called Qt Modelling Language (QML). Although this standard form already facilitates editing features, arrangement of attributes depends on their original order in the data source. For complex data models such as CityGML and the Energy ADE, further formatting is required to make feature editing more user-friendly and intuitive. QML can be used to customize an *Attributes Form* for this purpose. An example form is provided in Figure 13, attributes are presented in a more organised manner in contrast to the attribute table for the same layer in Figure 12.

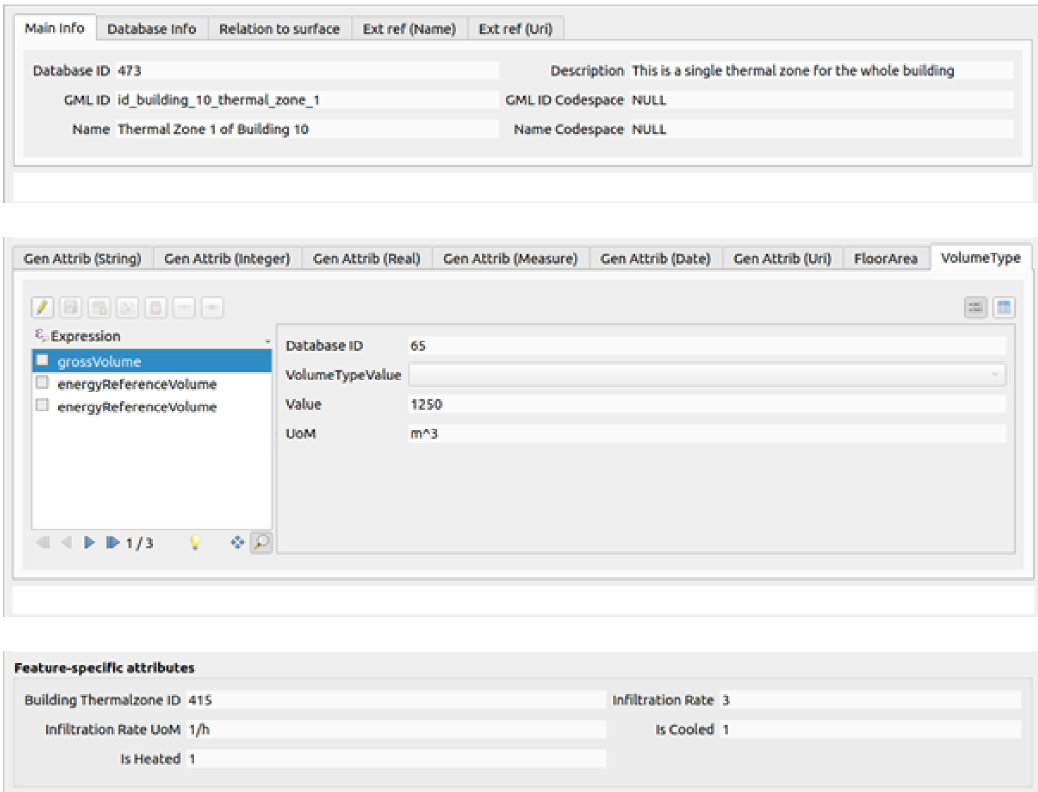


Figure 13: Customised Attributes Form for a QGIS layer.

2.2.3 Qt

Qt is a framework used to build Graphical User Interfaces (GUIs), that of QGIS serving as one example. Although it is native to C++, Python bindings are developed which combine "all the advantages of the Qt C++ cross-platform widget toolkit with Python, the powerful and simple, cross-platform interpreted language" (Willman, 2022), and allow developers to create Qt interfaces in Python. It provides Qt Designer, a toolkit which enables non-programmatic user interface design. In this research, Qt Designer is used to structure graphical user interface elements. Among a variety of GUI design styles, (Zanzoterra, 2018) reports *Qt Widgets* as the most common, stable and well-tested approach to Qt development. For this reason, *Qt Widgets* served as a starting point for further developing the 3DCityDB-Tools GUI.

GUI elements use *signals* and *slots* to communicate. *"A signal is emitted when a particular event occurs."* (The Qt Company, 2023b), clicking and scrolling are examples of these events. The Qt documentation further explains that an event triggers a function, also called a *slot*, to perform a task in response to a signal. Some classes contain their own signals and slots, but *Qt Widgets* allow developers to build custom data or message transfer channels. Hence, customised tasks can be connected to dialog elements.

According to the Qt documentation, a *QPushButton* is the most frequently used graphical user interface element. As the name leads on, it is a button which can be pressed to emit a *clicked* signal and *"command the computer to perform some action or to answer a question."* (The Qt Company, 2023a). An example of this button is given in Figure 14. A customised label can be assigned to a *QPushButton*, as well as to any other GUI element. In this context, the *QPushButton* is labelled *"Cancel"*. This is done using a *QLabel*.

Figure 14: A QPushButton with a QLabel.

A *QGroupBox* is another example of a *Qt Widgets* class to whose objects actions can be connected. It provides a frame in which other elements can be placed as child elements, and can be activated or deactivated using its *setChecked* method if it is checkable. Activating or deactivating it will result in the same action also being performed on its child elements. Figure 15 shows an example of this object with a few child elements although only its child elements can be checked in this instance.

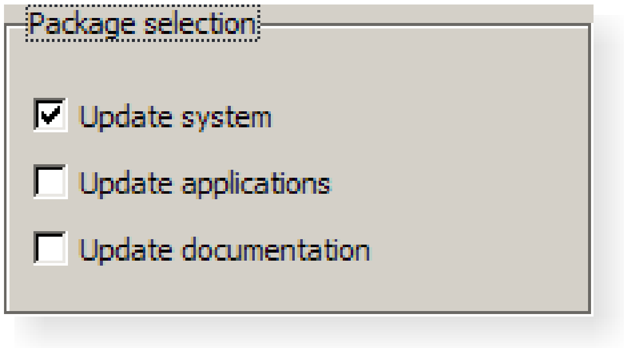
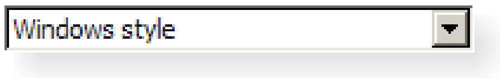
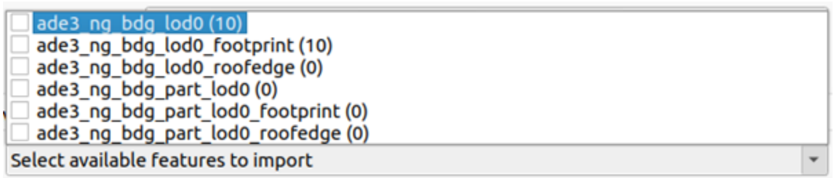


Figure 15: A QGroupBox.

A *QComboBox* allows displaying a list of options from which one item can be selected. To initiate tasks in response to a selection, custom slots can be developed and attached to its signals. It can be modified to enable selection of more than one item, as is done by its pyQGIS subclass *QgsCheckableComboBox*. Figure 16 shows examples of both elements.



(a) A QComboBox



(b) A QgsCheckableComboBox

Figure 16: Qt GUI elements that allow displaying a list of options from which a selection can be made.

2.2.4 Related QGIS Plug-ins

2.2.4.1 CityJSON Loader

Ledoux et al. (2019) present CityJSON, a developer-oriented data format designed to overcome the complexity and verbosity of GML as well as the difficulty of efficient web-based processing and exchanging of CityGML models. Although this study states that version 1.0.0 of CityJSON encodes CityGML version 2.0, the data format has been upgraded to version 1.1.3 which is tailored for CityGML 3.0. The one commonality is the encoding of only a subset of the data model, leaving out only those components seldom used in practice or that would unnecessarily complicate the encoding. A CityJSON file contains one JSON object of type ”CityJSON” and would typically contain the properties shown in Figure 17.

```
{
   "type": "CityJSON",
   "version": "1.0",
   "CityObjects": {},
   "vertices": [],
   "appearance": {}
}
```

Figure 17: Properties of a CityJSON object. (Figure adapted from (Ledoux et al., 2019))

CityObjects are organised into two levels. In the first are top-level classes which are typically named after their respective feature types, for instance *Building* shown in Figure 2. Part-named features such as *BuildingPart* and all other classes belonging to a CityGML module constitute the second level. Figure 18 gives a better illustration of this classification. CityJSON encodes both in a similar way inside the CityObjects dictionary of the CityJSON object as depicted in Figure 19. Storing them in this manner effectively unnests CityGML hierarchies.

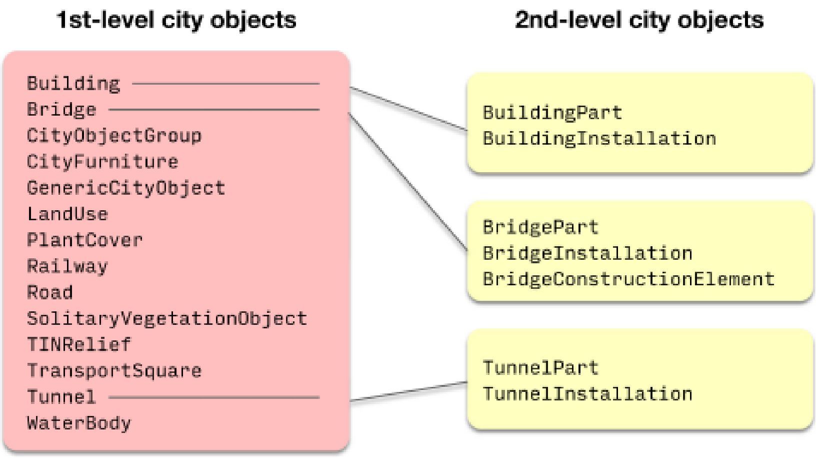


Figure 18: *The CityGML classes implemented in CityJSON (same names as CityGML classes) divided into 1st and 2nd level CityObjects.* Figure adapted from (Ledoux et al., 2019)

CityJSON implements support for geometries by defining 3D geometric primitives, MultiPoint, MultiLineString, MultiSurface, CompositeSurface, Solid, MultiSolid and CompositeSolid. Their storage in JSON objects by the *"geometry"* property of a *CityObject* is well-documented online at https://cityjson.org/specs. In addition, semantic surfaces introduced by CityGML from LoD2 are treated as JSON objects in the *"surfaces"* array which is placed in the geometry dictionary and linked to their respective vertices using the *"values"* array as Figure 20 summarises.

```
"CityObjects": {
   "id-1": {
       "type": "Building",
       "attributes": {...},
       "children": ["id-2", "id-3"],
       "geometry": [{...}]
   },
   "id-2": {
       "type": "BuildingPart",
       "parents": ["id-1"],
       "geometry": [{...}]
```

Figure 19: *Storage of "Building" inside the CityObject dictionary of a CityJSON object.* Figure adapted from (Ledoux et al., 2019)

```
{
    "type": "Solid",
    "lod": 2,
    "boundaries": [
        [ [[0,3,2,1,22]], [[4,5,6,7]], [[0,1,5,4]],
     [[1,2,6,5]] ]
    ],
    "semantics": {
        "surfaces" : [
            { "type": "RoofSurface" },
            {
                "type": "WallSurface",
                "paint": "blue"
            },
            { "type": "GroundSurface" }
        ],
        "values": [ [0, 1, 1, 2] ]
    },
}
```

Figure 20: *Storage of the geometric information of a CityObject by CityJSON object.* Figure adapted from (Ledoux et al., 2019)

For this encoding, the CityJSON Loader was developed *"with the intention of making CityJSON (and, thus, 3D city models) more accessible to researchers and practitioners through general-purpose GIS software (in this case, QGIS)."* (Vitalis et al., 2020). The plugin has one window (Figure 21) which allows users to create QGIS features with multi-polygon geometries that represent the objects of the city model under the assumption that every city object is composed of multiple surfaces. Several layer configurations make this possible. A CityObject itself as a multipolygon, its levels of detail or semantic surfaces as polygons can be mapped to QGIS features which can be further packaged as layers grouped according to the CityGML feature type or LoD. An alternative would be to combine both these packaging mechanisms such that a layer is created for every LoD of every module. However, this plugin does not have customised attribute forms for user-friendly feature editing neither does it support any ADE currently.

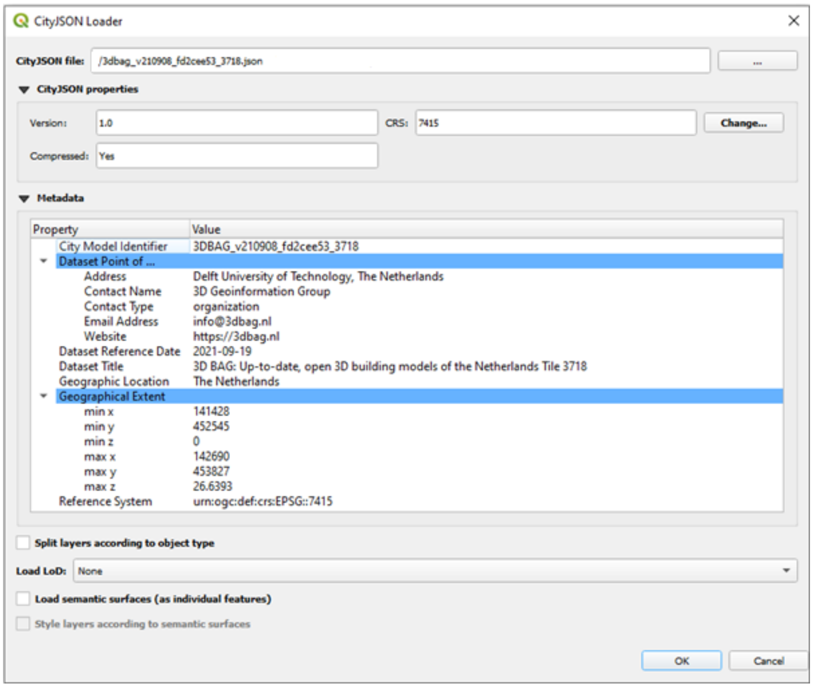


Figure 21: Dialog of the CityJSON Loader QGIS plugin.

2.2.4.2 3DCityDB-Tools for QGIS

The 3DCityDB-Loader (Pantelios, 2022), now renamed to 3DCityDB-Tools for QGIS, is a Python-and-Qt-based QGIS plug-in designed to simplify user interaction with CityGML data encoded in the complex 3DCityDB. It has a server-side component also called QGIS Package that consists of a database schema and a front-end whose 3 features are the QGIS Package Administrator Layer Loader and Bulk Deleter. Currently, the plug-in has no support for ADEs. Chapter 3 is dedicated to describing this plugin in greater detail to provide a better understanding for the reader.

3 3DCityDB-Tools for QGIS

3.1 Introduction

3DCityDB-Tools for QGIS is a QGIS plug-in developed to to "facilitate the use of 3DCityDB for users of different fields and expertise with the common denominator being the well-accustomed QGIS environment" (Pantelios, 2022). Its server-side component, QGIS Package, provides user and layer management capabilities to the client although it can be used independently and by other external applications. The client gives users access to these functionalities using an easy-to-use graphical user interface which does not necessarily require advanced knowledge of the complex 3DCityDB environment described in Section 2.2.1.

3.2 Back-End

3.2.1 Defining a Layer

CityGML modules contain features that are children of the *CityObject* class, have at least one associated geometry type and specified levels of detail. One example of a class with these characteristics is *_AbstractBuilding* in Figure 2. It has 3 geometry types, *gml::_Solid*, *gml::MultiSurface* and *gml::MultiCurve* whose levels of detail vary between 0 and 4. Specifically, *gml::_Solid* has four levels of detail, *lod1Solid* to *lod4Solid*. Though *_AbstractBuilding* is an abstract class and cannot be instantiated, it is inherited by its subclasses *Building* and *BuildingPart*.

Section 2.2.1 details the complexity of the 3DCityDB, how attributes and geometry of a feature are stored in multiple tables. Therefore, in compliance with the SFM implemented by QGIS, each instance of *Building* in a table must have attribute columns and a geometry property. This structure exactly describes what 3DCityDB-Tools refers to as a *Layer*. To carry forward the scenario of *Building* and *gml::_Solid*, each of the 4 levels of detail is treated as a unique geometric representation of *Building* features. Hence, 4 corresponding tables, reserved for each level of detail, with one geometry column are created such that *Building* features have their attributes and a spatial property as well in the same table. Each of these 4 tables would then stand as a *Layer*. Since all CityGML features supported by 3DCityDB-Tools currently have at least one geometry and level of detail, they produce layers of type *VectorLayer*.

3.2.2 Creating a Layer

As aforementioned, a layer is simply a database table with attribute columns and a geometry property. It is created in 2 steps. First, SQL statements to update the layer metadata, gather attributes and geometry from different 3DCityDB tables, and make the layer updatable are generated in one function. Adopting the example of *Building* and *gml::_Solid*, a separate layer is created for each of the 4 levels of detail. For each one, a Materialized View containing *id* and *geometry* columns is created. Materialized views are used for efficiency reasons as "...querying the geometry table on the fly would take a lot of time, so from several experiments we have decided to pre-generate the geometries" (Pantelios, 2022). The *id* column is later used to join to all geometries to their corresponding *Building* attributes to form a View, which is actually a layer as defined in Section 3.2.1.

Following this, SQL statements that attach triggers to the layer are also generated. A function named *generate_sql__triggers* is called and used to draft SQL code that attaches triggers to the layer. It requires a user schema, layer name and trigger function suffix to dynamically generate and attach a trigger function to a given layer. The function iterates over the INSERT, UPDATE and DELETE commands and builds a SQL statement that creates a trigger which in turn prescribes a trigger function to be subsequently executed for the relevant data query operation. Triggers redirect a query to an appropriate trigger function that prevents the insertion of new records into the tables on which a layer is built or allows an existing record to be updated or deleted.

Next, these SQL statements are executed in another function, resulting in:

1. removal of the metadata record for that layer, if any, from the layer metadata table.
2. (re-)creation of the layer.
3. insertion of a new metadata record for the (re-)created layer in the layer metadata table.
4. creation of triggers that are fired in the event of an INSERT, DELETE or UPDATE operation on the layer.
5. linking triggers to the trigger functions which they invoke when INSERT, DELETE or UPDATE operations are performed on the layer.

3.2.3 Layer Metadata

The layer generation process also involves creating a new metadata record of the layer which is then inserted in the layer metadata table. Information about the layer that is captured includes:

- 3DCityDB schema.

  The 3DCityDB schema which contains the tables used to create the layer.

- ADE prefix.

  If the layer is for an ADE feature, the ADE prefix takes the value which is specified when the 3DCityDB is extended with the ADE. In the current state of the plugin, no ADE is supported, therefore a *NULL* value is stored.

- Layer type.

  This is the layer type produced from the feature. Currently, it takes the value *VectorLayer* for all layers generated by the plugin.

- Feature type.

  The CityGML module under which the feature falls, if any.

- Top-level class.

  Top-level CityGML class which the feature inherits, if any.

- Class name.

  Name of the class whose metadata is in the current record.

- Level of detail.

  Level of detail for the geometric representation of the layer.

- Layer name.

  Name of the layer.

- Materialized view name.

  Name of the materialized view containing layer geometry.

- Attribute view name.

  This is a redundant column originally intended to store the layer name.

- Number of features.

  Number of features in the layer.

- Creation date.

  Date on which the layer was created.

- Refresh date.

  Latest refresh date of the layer's materialized view. This property is only used for layers with geometry, otherwise it is *NULL*.

- QML attribute form name.

  Name of the customised attribute form for the layer.

- QML symbology form name.

  Name of the customised layer symbology form.

- QML 3D symbology name.

  Name of the customised layer 3D symbology form.

- Layer columns associated to an enumeration.

  This property stores an array of arrays containing table and enumeration column pairs.

- Layer columns associated to a codelist.

  This property stores an array of arrays containing table and codelist column pairs.

3.2.4 Updating a Layer

The plug-in only permits layer updates made using DELETE and UPDATE queries. Furthermore, execution of these operations directly on the layer is prevented. This is because a layer is composed of information from different 3DCityDB tables, and does not satisfy the one of the conditions required by PostgreSQL for a view to be directly updatable. *"The view must have exactly one entry in its FROM list, which must be a table or another updatable view."* (The PostgreSQL Global Development Group, 2023b), which is not the case as previously described for layers produced by 3DCityDB-Tools. Instead, trigger functions are used as a starting point in a chain of functions that simulate the effect of an updatable view.

When an INSERT query is attempted on a layer to introduce new records into the view, the trigger for this operation is fired and in turn invokes the trigger function. Rather than completing the insertion, the trigger function raises an error which notifies the operator that QGIS Package does not permit this action. A DELETE query similarly sets off a trigger which then points to a trigger function. Subsequently, the trigger function invokes a 3DCityDB function, named as *del_cityobject*, which deletes a *CityObject* from the database and any associated hierarchical information in other tables. For example, to remove a *Building* object from the 3DCityDB, its *id* is given to the *del_building* function which erases information from the *cityobject*, *building* and *surface_geometry* tables.

For an UPDATE query, several functions are chained to the relevant trigger function linked to a layer. Now, the first step in creating the effect of an updatable view involves separating attributes intended for different 3DCityDB base tables from the query. This action is performed inside the trigger function using the *Type* construct provided by PostgreSQL which *"...registers a new data type for use in the current database."* (The PostgreSQL Global Development Group, 2023a). QGIS Package creates a *Type* for each 3DCityDB table such that it contains the same attributes as the table and behaves akin to a class in object-oriented programming languages like Python. For each layer produced by the plugin, the associated types are known beforehand. In the event of an UPDATE, each trigger function in the back-end is hardcoded to assign attributes in the query to their corresponding types. Following this is a *View Update Function*. It receives all objects from the trigger function. Thereafter, it hands over each object to a *Table Update Function* which actually performs the update on the table represented by the type which it receives. Suppose an UPDATE query is performed on a layer for the *Building* class, the trigger halts the operation. The trigger function takes over, separates attributes from the query into 2 objects that capture *CityObject* and *Building* attributes, and hands them over to the view update function. From here, two other functions receive each of the objects and carry out updates to the *cityobject* and *building* relations.

3.3 Front-End

3.3.1 QGIS Package Administrator

QGIS Package Administrator, shown in Figure 22 is the window used for installing or uninstalling QGIS Package and managing users. By installing QGIS Package to the database first connected to, the *qgis_pkg* schema is created in the back-end together with tables and functions for user and layer management. Uninstalling in turn removes the schema from the database. User management entails creating or removing a separate database schema for a user which will contain all layers they create, and assigning them privileges that either restrict them to only accessing data or enable both viewing and modifying database tables. For a user to appear in this dialog so that they may be assigned a schema and privileges, they must be created in the database beforehand. Typically, only database administrators or users with sufficient privileges can perform these tasks.

Among the tables that are created when QGIS Package is installed are those that store enumeration and codelist information as well as a template table for layer metadata. These are *enum_lookup_config*, *enumeration_template*, *enumeration_value_template*, *codelist_lookup_config_template*, *codelist_template*, *codelist_lookup_config* and *layer_metadata_template*. The template metadata table is identical to the table discussed in Section 3.2.3. What differentiates the two tables is that *layer_metadata_template* is kept in the *qgis_pkg* schema and is not used to capture any information. Instead, it is used to create replica tables in user schemas which actually capture the layer metadata detailed in Section 3.2.3. Hence, the metadata table will not be discussed further.

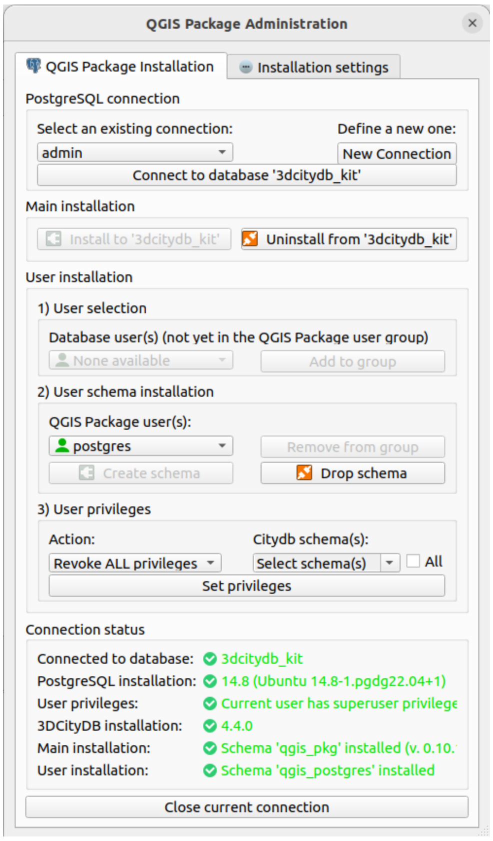


Figure 22: QGIS Package Administrator dialog.

For the purpose of illustrating how enumeration tables are structured, *RelativeToWaterType* (Figure 3b) will be used together with the *CityObject* class. In addition, only those columns required for explanation will be presented. The table *enum_lookup_config* contains configuration information for enumerations in the CityGML data model as Table 1 demonstrates. Next, *enumeration_template* records contain a data model, enumeration name and namespace in the *data_model*, *name* and *name_space* columns respectively. A row representing *RelativeToWaterType* in this table would have "CityGML", "RelativeToWaterType" and the uniform resource locator "https://schemas.opengis.net/citygml/2.0/cityGMLBase.xsd" as values in the columns given above. Actual enumeration values are stored in the table *enumeration_value_template*. Carrying on with *RelativeToWater* and one of its values "entirelyAboveWaterSurface", this value is stored under the column *value* and its description "(City)Object entirely above water surface" under *description*. This table also contains a foreign key to *enumeration_template* which links an enumeration value to its enumeration name and data model. Tables 2 and 3 show examples of *enumeration_template* and *enumeration_value_template* using the given information.

The codelist containers in *qgis_pkg* are identical to their enumeration counterparts, except they are for codelists. Columns of *codelist_lookup_config_template*, *codelist_template* and *codelist_value_template* are the same as those of *enum_lookup_config*, *enumeration_template* and *enumeration_value_template* respectively. Using the *class* property of the *Building* feature in CityGML and one of its codelist values "1120" together with the description "Healthcare" for illustration, Tables 4, 5 and 6 show values that go into each of the 3 relations.

In addition to these tables, two views are created when QGIS Package installation is performed. These are *v_enumeration_value_template* and *v_codelist_value_template*. The first view is formed as a join between *enumeration_template* and *enumeration_value_template* with the columns *data_model*, *name*, *value*, *description* and *name_space*. The second view has the same structure, but from a join between *codelist_template* and *codelist_value_template*. Using the previous examples, *RelativeToWaterType* and _AbstractBuildingClass, Tables 7 and 8 show how these views are structured.

Aside from QGIS Package installation, QGIS Package Administrator is also used for user management. This involves creating or removing a user schema, and assigning or revoking database privileges. When a user schema is created, the template layer metadata table is copied to this schema, and so too are the views *v_enumeration_value_template* and *v_codelist_value_template*. However, all three relations are renamed to *layer_metadata*, *v_enumeration_value* and *v_codelist_value* respectively. All metadata is kept in the *layer_metadata* table in the user schema and not in the *qgis_pkg* schema. Having enumerations and codelists in the user schema allows for them to be linked to their associated CityGML feature attributes when the user imports layers into QGIS.

Table 1: An example of the table enum_lookup_config using *CityObject* and *RelativeToWaterType*.

ade_prefix	source_class	source_table	source_column	target_table	key_column	filter_expression
NULL	CityObject	cityobject	relative_to_water	v_enumeration	value	data_model = CityGML 2.0 AND name = RelativeToWaterType

Table 2: An example of the table enumeration_template using *CityObject* and *RelativeToWaterType*.

id	data_model	name	name_space
2	CityGML	RelativeToWaterType	https://schemas.opengis.net/citygml/2.0/cityGMLBase.xsd

Table 3: An example of the table enumeration_value_template using *CityObject* and *RelativeToWaterType*.

id	enum_id	value	description
6	2	entirelyAboveWaterSurface	(City)Object entirely above water surface

Table 4: An example of the table codelist_lookup_config_template using the CityGML *Building* class.

ade_prefix	source_class	source_table	source_column	target_table	key_column	filter_expression
NULL	Building	building	class	v_codelist	value	data_model = CityGML AND name = _AbstractBuildingClass

Table 5: An example of the table codelist_template using the CityGML *Building* class.

id	data_model	name	name_space
4	CityGML	_AbstractBuildingClass	https://www.sig3d.org/codelists/standard/building/2.0/_AbstractBuilding_class.xml

Table 6: An example of the table codelist_value_template using the CityGML *Building* class.

id	code_id	value	description
40	4	1120	Healthcare

Table 7: An example of the view v_enumeration_value_template using *CityObject* and *RelativeToWaterType*.

id	data_model	name	value	description	name_space
6	CityGML 2.0	RelativeToWaterType	entirelyAboveWaterSurface	(City)Object entirely above water surface	https://schemas.opengis.net/citygml/2.0/cityGMLBase.xsd

Table 8: An example of the view v_codelist_value_template using the CityGML *Building* class.

id	data_model	name	value	description	name_space
40	CityGML 2.0	_AbstractBuildingClass	1120	Healthcare	https://www.sig3d.org/codelists/standard/building/2.0/_AbstractBuilding_class.xml

3.3.2 Layer Loader

The Layer Loader dialog is used to create, refresh and drop layers, or load them into QGIS for visualisation or attribute editing. This is arguably the window users mostly interact with. It has 3 tabs which will be discussed in the following paragraphs.

3.3.2.1 User Connection

In the User Connection tab (Figure 23), users can create, refresh or drop layers. First a connection is made to a particular 3DCityDB schema. Next, at least one specific feature type (CityGML module) can be selected, and all its associated layers can be created. If no selection is made, layers for all feature types present in the loaded 3DCityDB schema will be created when "Create layers for schema '{sch}'" is clicked. Doing so emits a signal which is caught by the *evt_btnCreateLayers_clicked* slot, leading to initiation of a separate thread outside the main event loop of the plugin. The *QThread* object is handed a *QObject* worker class, *CreateLayersWorker*, whose *create_layers_thread* method is connected to the *started* signal of the thread and responsible for invoking server-side functions that create layers for each feature type. For a given class in a CityGML module, its layer is created as described in Section 3.2.2.

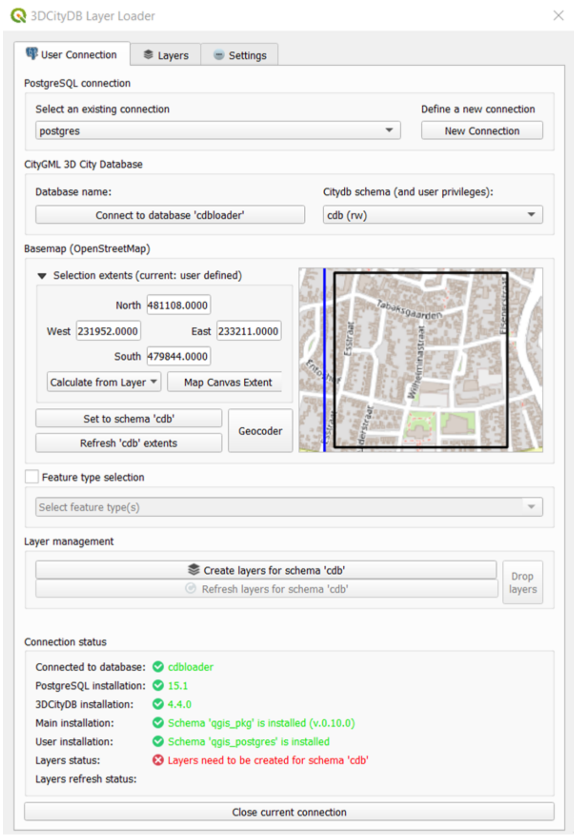


Figure 23: The User Connection tab in the Layer Loader.

Refreshing layers follows similar steps to creating. Clicking the button "Refresh layers for schema '{sch}'" emits a signal which is caught by the *evt_btnRefreshLayers_clicked* slot, leading to another thread being created by the function *run_refresh_layers_thread*. The thread is handed a *QObject* worker class, *RefreshLayersWorker*, whose *refresh_all_gviews_thread* method obtains materialized views to be refreshed from the *layer_metadata* table in the user schema when the thread is started. Likewise, clicking "Drop layers" emits a signal which is caught by the *evt_btnDropLayers_clicked* slot, leading to another thread in the function *run_drop_layers_thread*. The *QThread* object is handed a *QObject* worker class, *DropLayersWorker*, whose *drop_layers_thread* method is connected to the *started* signal of the thread and responsible for deleting layers and their metadata from the user schema. To illustrate, Figure 24 generalises the sequence of events after clicking a button in the User Connection tab to create, refresh or drop layers.

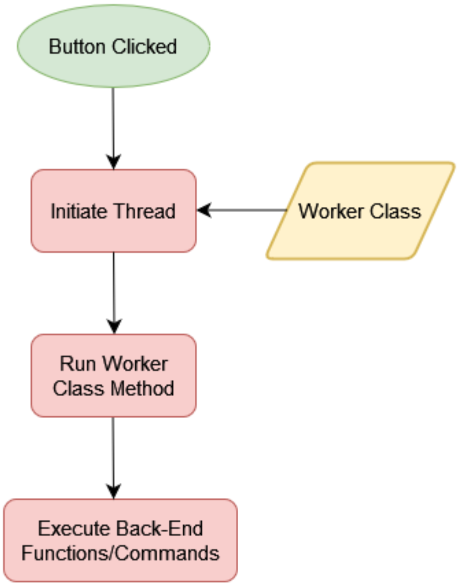


Figure 24: Workflow followed in creating, refreshing and dropping layers through the User Connection tab of the Layer Loader using a thread and a method in a worker class.

3.3.2.2 Layers

In the Layers tab (Figure 25), users can select a feature type and level of detail, and subsequently choose which layers to import into QGIS among those that meet the specified criteria. For example, *Building* and LoD4 could be selected. As layers are pulled from the database, they are also linked to their respective enumerations in the user schema. Doing so enables these standardised values to appear in drop-down menus in the layer *Attributes Form*. Figure 26 illustrates the effect of this linkage, for the *RelativeToWaterType* enumeration, in a form. It is also at this point that layers are linked to their QML files "...to stylize the symbology (only color) and the attribute form." (Pantelios, 2022).

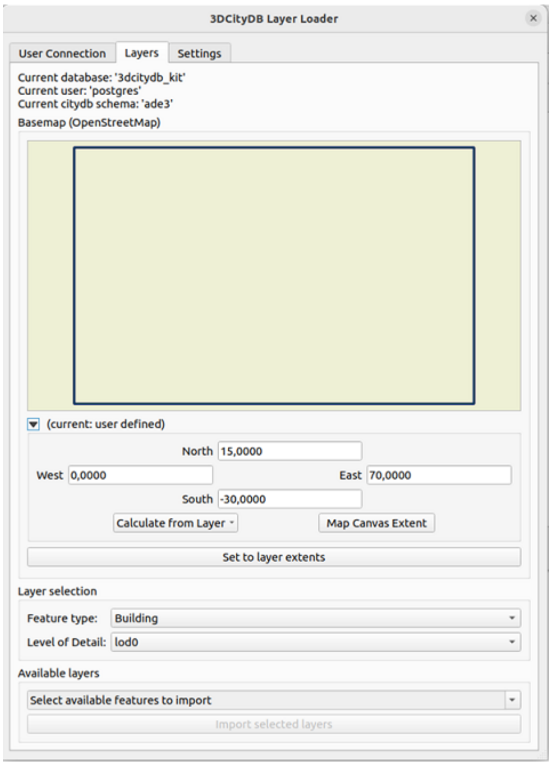


Figure 25: The Layers tab in the Layer Loader.

Once layers have been imported into QGIS, they are sorted in a tree in the *Layers* panel. A database name, schema name, as well as CityGML feature type and level of detail are the parameters used in creating tree nodes for organising the layers. An example is given in Figure 11.

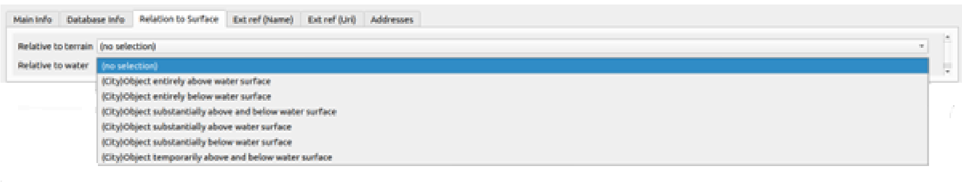


Figure 26: Options for the *RelativeToWaterType enumeration in a* Building *layer.*

3.3.2.3 Settings

In the Settings tab (Figure 27), users have the options to perform geometry simplifications settings and control the maximum number of features to import from the database. Also, users can select codelist sets to load, though those CityGML are loaded by default. This enables them to appear in drop-down menus in the layer *Attributes Form*.

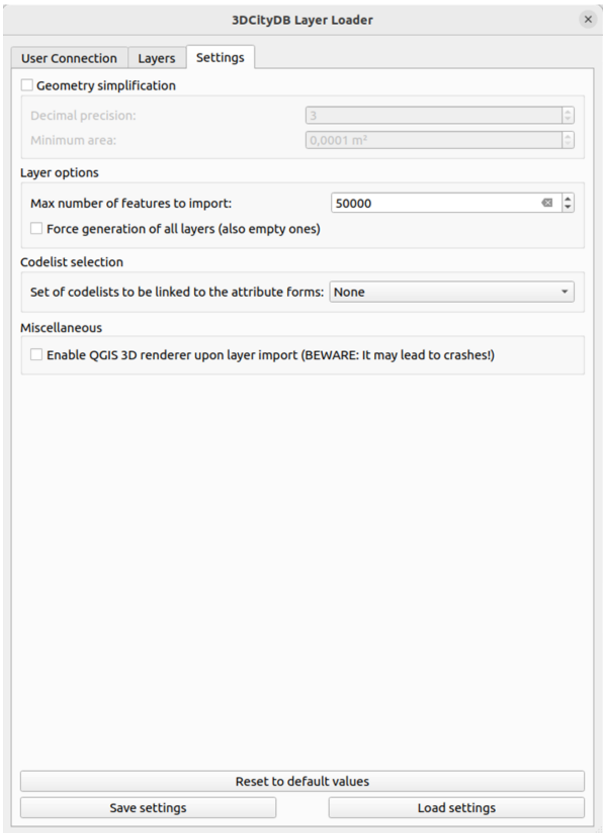


Figure 27: The Settings tab in the Layer Loader.

3.3.3 Bulk Deleter

To delete features in their entirety (geometry, topology, semantics and properties) from a 3D City Database schema is the purpose of the Bulk Deleter, as is inherent in the name. While the dialog in Figure 28 has been part of the plugin since version 0.7, it will be discussed further in Section 4.2.3.3 due to its connection to objective 5 of this study.

Figure 28: The Bulk Deleter dialog.

4 Extending 3DCityDB-Tools with ADE Support

4.1 Method

4.1.1 Introduction

Addition of ADE support to 3DCityDB-Tools was tailored to maintain the current architecture and user experience of the plugin. Reusing code as much as possible and minimising changes in the user interface guided the above requirements. Furthermore, this research set out to develop a generic approach for supporting CityGML ADEs though using the Energy ADE as a test case. For both the back-and-front ends, the following sections report the strategies followed and how they were implemented.

4.1.2 Incremental Development

To mitigate integration and architectural risks, this development approach breaks down a project in one of several ways. Figure 29 depicts the underlying concept. A product can be developed as a series of subsystems, each one having its own lifecycle. In another form, incremental development first defines overall requirements and then proceeds to implementation in iterations (Figure 30). This allows frequent feedback loops and easy isolation and handling of issues. In this way, event-driven systems such as the 3DCityDB-Tools GUI can be developed. Changing requirements are accommodated, and integration issues are mitigated quickly. This is particularly important in the context of CityGML and the Energy ADE due to a complexity in these data models acknowledged by several studies in literature. Their hierarchies are factored in with each increment.

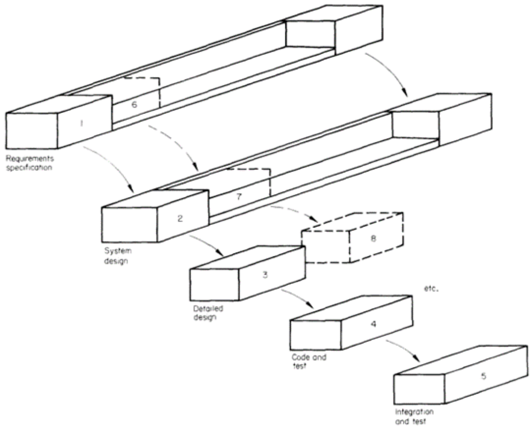


Figure 29: *"Framework incremental."* Source: (Graham, 1989)

In line with this approach, the back-end was first enhanced followed by the front-end. The goal was to have a complete server-side with all functionality required by the user interface before extending it with ADE-aware elements and operations. Not only did incremental development guide further development of 3DCityDB-Tools from this rather wide picture, it also influenced development at a granular level. On the server-side, ADE support was facilitated by files that were compiled one after another. Development of units in each file took an incremental build and test strategy which involved implementing and testing until the desired performance was achieved. Similarly, ADE capabilities were added to the ADE element by element and function by function. Concurrent development and testing was also followed to ensure that a ADE component triggered the correct events, traced the intended execution path and performed the desired action.

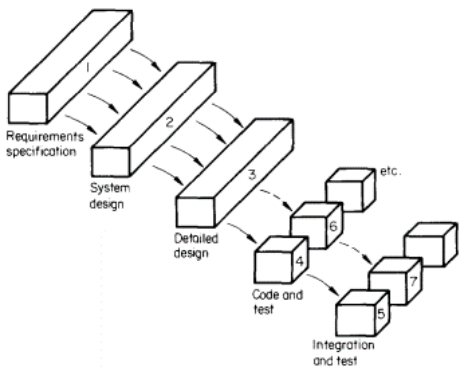


Figure 30: *"Incremental build and test."* Source: (Graham, 1989)

4.1.3 Iterative Development

The goal of this approach is effectiveness, speed is at the moment overshadowed by the need to have a product which does what it is intended to do. Hence, Iterative Development involves decomposing a project to enable fast development through prototyping. With respect to CityGML and the 3DCityDB, adoption of this approach complemented incremental development.

The addition of the Energy ADE introduces deeper hierarchies in a CityGML dataset, and greater relational intricacy in a 3DCityDB instance. For this reason, 10 classes packaged in different modules but representative of the Energy ADE KIT Profile were picked as an initial test set in investigating further development of 3DCityDB-Tools. *_AbstractBuilding*, *ThermalZone*, *ThermalBoundary*, *ThermalOpening*, *WeatherStation*, *WeatherData*, *EnergyDemand*, *DailySchedule*, *PeriodOfYear* and *RegularTimeSeries* served as a starting point as they presented various dissimilarities in LoDs, inheritance levels and nature of associations with other classes including those native to CityGML. For instance, the KIT Profile class *_AbstractBuilding* inherits CityGML *_AbstractBuilding* and by extension has an association to *gml::_Solid* and *gml::MultiSurface* which possess its spatial properties. This means it can be represented at 4 LoDs for *gml::_Solid* with *lod1Solid* up to *lod4Solid* geometry, and at 5 LoDs for *gml::MultiSurface* by *lod0FootPrint* and *lod0RoofEdge* as well as *lod1MultiSurface* to *lod4MultiSurface*.

Whereas, the other 9 class do not all have an associated geometric representation. Those of the 9 that do have no level of detail specified in the data model, as is the case for *ThermalZone*. It has a *volumeGeometry* property of type *GM_Solid* but the data model does not specify any levels of detail for the spatial representation of a *ThermalZone*. *DailySchedule*, *EnergyDemand*, *PeriodOfYear* and *WeatherData* do not descend from any other class unlike the other 6 which take after *CityObject* or *_AbstractBuilding* in CityGML if not a superclass in the same KIT Profile module. Taking this route was a measure to reduce the complexity of the Rubik's cube at hand while developing solutions that generalise to all classes. A second iteration incorporated the remaining ADE classes.

4.1.4 Back-End: Rethinking the Layer Concept

Currently, the 3DCityDB-Tools only produces layers of type *VectorLayer* which contain information about CityGML features that have a spatial representation. For each supported class, every associated level of detail for a given aggregate or composite geometry produces a layer containing its attributes. Sections 3.2.1 and 3.2.2 explain how these layers are defined and created respectively.

However, the ADE adopted in this research does not directly fit into this template. It brings about different elements which necessitate the introduction of a layer taxonomy for layer generation, metadata and structuring in the QGIS layer tree. These elements are listed below with examples:

- new "child" CityObjects which do not have geometry (*UsageZone*).
- new "child" CityObjects which have geometry (*ThermalBoundary*).
- classes which may have geometry but no defined levels of detail (*ThermalZone*, *Facilities*).
- extended CityObjects (*_AbstractBuilding*).
- new top-class CityObjects (*WeatherStation*).
- non-CityObjects which have geometry (*WeatherData*).
- non-CityObjects which do not have geometry (*EnergyDemand*).

An adapted layer classification is therefore put forward for a 3DCityDB-Tools extended with the Energy ADE KIT Profile. In addition, the absence of defined LoDs in the KIT Profile is a void filled by assigning a new Level of Detail (LoD) termed *LoDX* or *lodx*.

New and extended *CityObjects* are labelled *VectorLayer* since they have geometry. This only stretches the current definition of this layer type which models a *CityObject* with spatial properties and at least one defined level of detail. An ADE class represented by a *VectorLayer* is commonly mapped to three 3DCityDB tables, *cityobject*, *ng_cityobject* and another which takes the class name. *ThermalZone* for instance produces a *VectorLayer* with attributes from *cityobject*, *ng_cityobject* if any and *ng_thermalzone*. Such a layer will actually be a *View* built on a *Materialized View* containing geometry, with non-spatial properties from the three tables listed. Another example is that of *_AbstractBuilding*, though it is an exception because it extends an existing *CityObject* and can have attributes stored in four tables. These tables represent the CityGML *_CityObject* and *_AbstractBuilding* classes, as well as their respective subclasses *_CityObject* and *_AbstractBuilding* in the Energy ADE. A layer produced from this class will also be a *View* that extracts geometry from a *Materialized View* as detailed in Section 3.2.1, but with attributes from the four tables listed.

CityObjects that are not associated with any geometry in the data model are designated as *VectorLayerNoGeom*. Their information is similarly extracted from three tables, except these classes are *CityObjects* with no geometric representation. As an example, *Facilities* is an underlying class for a *VectorLayerNoGeom* with attributes from *cityobject*, *ng_cityobject* if any and *ng_facilities*. Its layers will be views that capture only attributes from the mentioned relations.

A class which is not a *CityObject* but may have a geometric representation is a *DetailView*. A *DetailView* layer will have some commonalities with those of type *VectorLayer*. Though a KIT Profile class from which a *DetailView* will be created is typically not a *CityObject*, the 3DCityDB stores some information about the feature in *cityobject* and another table named after the class itself. This is the case for *WeatherData*, whose table is *ng_weatherdata*. Therefore, its layers will also be views that borrow columns from the referenced relations and from a geometry view containing *gml::Point* geometry.

Lastly, a *DetailViewNoGeom* signifies a KIT Profile class which neither has geometry nor inherits *CityObject*. Some layers of this type merge information from at least two tables, including *cityobject* and another named after class with the *type* stereotype. A fitting example is that of *EnergyDemand* whose layers extract attributes from *cityobject* and *ng_energydemand*. In the case of inheritance from an abstract class, *DetailViewNoGeom* layers will pull attributes from three tables, the third being that of the abstract class. The relations *cityobject*, *ng_timeseries* and *ng_regulartimeseries* provide attributes for a *RegularTimeSeries* layer, considering that class as an example. Other layers of this type are created from attributes taken from a single table, as is the case for classes like *VolumeType* and *FloorArea*.

Figure 31 summarises the thinking behind these four layer types. All layer types laid out in this section are based on identification of linked

3DCityDB tables and extracting relevant data from them into a view into a view. Assessing from the above, the choice of which layer type to designate for a given Energy ADE KIT Profile class is influenced by 3 pieces of information - presence or absence of geometry, relationship of ADE classes to those in CityGML and encoding of ADE classes in the 3DCityDB.

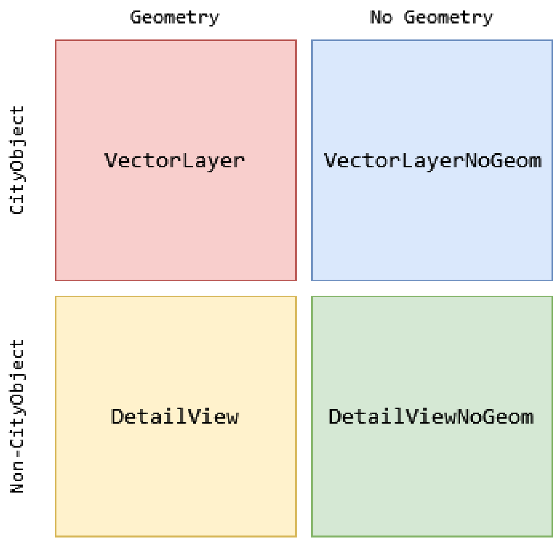


Figure 31: Layer classification.

4.1.5 Front-End: Enabling ADE Support

One of the uses of the QGIS Package Administrator dialog is installing the *qgis_pkg* schema in the back-end for CityGML functionalities only. This research aimed to have a QGIS Package installation process which also brings ADE capabilities to the front-end. In this way, users would then be able to create, refresh, drop and load ADE-aware layers within QGIS. However, a design decision was made to allow these actions to be carried out either for plain CityGML layers or for those extended by only one ADE at a time. Consequently, for a given 3DCityDB schema, the layer metadata table will only have layers for the CityGML base profile, or those associated with an ADE. The reason for this decision comes down to reducing complexity of the task at hand to quickly advance investigation of the research question and have a minimum viable product that supports an ADE. In addition, duplication of information is prevented. Taking an example of the *_AbstractBuilding* class in CityGML, its attributes will also be present in a layer created for its subclass

_*AbstractBuilding* in the Energy ADE KIT Profile. Hence, nearly the same data would be contained in two different layers if the above decision was not enforced.

With this also comes a requirement for attribute forms to be created, which are loaded when layers are imported into QGIS. Users are only exposed to a single attribute form, giving the impression that they are interacting with one database table. Though this is partially true, the previous section explains the relational complexity hidden by these layers. Having tailored forms for feature attribute editing additionally makes performing updates to ADE and CityGML tables more user-friendly by eliminating the need for broad technical knowledge of the 3DCityDB environment.

4.1.6 Tools and Data

Multiple tools were required to explore further development of 3DCityDB-Tools. PostgreSQL is the database management system in which test instances of the 3DCityDB version 4.4 were installed. For this, platform dependent scripts that are distributed as part of the 3DCityDB Suite were used on Ubuntu 22.04 Long Term Support. To import CityGML datasets, the 3DCityDB Importer/Exporter was employed. It also provides functionalites to transform the Energy ADE KIT Profile XML schema definition into relations, followed by extension of a 3DCityDB schema in the database. Python 3.9, complemented by the Qt framework for User Interface (UI) design, QGIS' Desktop version 3.22 and Application Programming Interface written in the same language, was the plugin development language for implementing and testing functionality. These tools were run on the Ubuntu 22.04 Long Term Support operating system, although all modifications maintain the platform-independent status of the plugin.

An artificial dataset was used in conducting this research. It was selelcted for having information for all but two classes in the the Energy ADE KIT Profile, and an unavailability of real-world datasets that are open-source.

4.2 Implementation

4.2.1 Introduction

Extensions of the CityGML data model introduce new elements into the `3DCityDBTools` frame. All packages in the Energy ADE KIT Profile put in place supplementary classes and attributes. To accommodate them in QGIS Package, database types, tables and views, functions, and triggers were devised in a set of scripts. This structure aligns with the current version of the plug-in and permits standalone use of QGIS Package or in combination with the plug-in.

4.2.2 Back-End

4.2.2.1 Creating a Layer

At initial development in the work of Pantelios (2022) , 3DCityDB-Tools only supported layers of type *VectorLayer* built on top of the CityGML base profile in which all *CityObjects* are associated with at least one geometry type. However, the Energy ADE brings about different elements which necessitated the introduction of a layer taxonomy elucidated in Section 4.1.4. The workflow for creating layers is similarly carried out in two distinct steps as discussed in Section 3.2.2. A few differences arise for the Energy ADE and will be explained in this section, but the general approach is the same.

Each CityGML feature type extended or introduced by this ADE has its own file containing a SQL generator function for every one of its classes. Benner (2018) elaborates that the Energy ADE extends the *Building* module of CityGML, but it also introduces *WeatherStation*. These functions that generate SQL code, which is later executed by another function, gather several pieces of information needed to create a layer and its metadata as SQL statements. This is done dynamically from within the second function which runs the SQL string by taking advantage of a uniform signature adopted by SQL-generating functions. The function signature incorporates:

1. a function name structured as *qgis_pkg.generate_sql_layers_ng_xx*.

   The suffix takes the name of a KIT Profile class in lowercase, for example *thermalopening*.

2. the same parameters - *usr_name*, *cdb_schema*, *perform_snapping*, *digits*, *area_poly_min*, *bbox_corners_array* and *force_layer_creation*.

These parameters represent a database user name, 3DCityDB schema, boolean value indicating if geometries should be snapped, number of digits to use for snapping, the minimum polygon area for snapping, geometry representing corners of a bounding box and a boolean value to determine if creating layers should be forced.

Which information is brought together and how depend on the type of layer for which SQL statements are to be generated. Generally all functions that produce SQL string begin by declaring variables used in the function and carrying out a few checks, then they prepare statements for:

1. erasing existing metadata of the ADE feature, and entering a new record in the layer metadata table,
2. counting the number of objects within a given geographic space (if the feature has an associated geometry),
3. joining information from different 3DCityDB tables,
4. attaching triggers to the layer, and
5. adding actual metadata to its previously prepared statement.

Among the declared variables are codelists and enumerations of a class, if any are stipulated by the data model. They are declared as nested arrays. Two values, table and column names, are stored in each inner array for both codelists and enumerations. Extending CityGML with the Energy ADE KIT Profile invites numerous codelist and enumeration entrants into the 3DCityDB-Tools framework. In the given order, *CurrentUseValue* and *WeatherDataTypeValue* can serve as examples of a codelist and an enumeration. The aforementioned checks verify the existence of the user name and 3DCityDB schema in the database. Following this each generator function creates a SQL statement to delete, if any, the existing record for its feature from the layer metadata table. A partial INSERT query into the same table with the columns whose metadata values will be later added is then appended to the SQL string. The next step is where differences emerge in the construction of SQL statements that fetch information from at least one location for the varying layer types.

For a KIT Profile feature whose layer is of type *VectorLayer* or *DetailView*, a *WHERE* clause is constructed and used as a spatial filter to count the number of its instances that are within a given geographic area. This is in fact a bounding box passed to the *mview_bbox* parameter of the function.

By joining the 3DCityDB table of the feature with *cityobject* on their *id* columns, the spatial filter can be applied on the *envelope* property of the latter table. Having this filter in place guarantees that only objects in the area of interest are added to the layer. If at least 1 record passes through the filter, some SQL statements are forged that capture the geometries of the feature instances within the bounding box into a *Materialized View*. Furthermore, all possible options for geometric representation for each class are exhausted to make certain that at least one geometry type is picked out for the layer for a given LoD. Taking *ThermalZone* as an example, geometry for its layers can be a *gml::_Solid* extracted directly from the *surface_geometry* relation, or *gml::MultiSurface* composed of bounding surfaces first identified through a join between *ng_thermalzone* and *ng_thermalboundary* on their *id* and *thermalzoneboundedby* columns. Solid geometry always takes precedence, thus the function for *ThermalZone* only takes the second route if the *volumegeometry_id* property in its 3DCityDB table is *NULL*.

Subsequently, the function that generates SQL code then fabricates statements that join geometry in the *Materialized View* beside associated attributes of the feature into a *View* using a common *id*. This means that for a *VectorLayer*, and where possible for a *DetailView* as well, the layer to be created will have attributes from the relevant 3DCityDB table named after the feature and from *cityobject*. This is the case for the for the *WeatherData* class which produces a *DetailView*. All its properties are acquired from two tables, *cityobject* and *ng_weatherdata*, including *gml::Point* geometry. While the above also applies to a *VectorLayer*, it can go as many as two steps further due to classes in the Energy ADE that inherit others in CityGML. Firstly, the *_CityObject* class in the Energy ADE inherits *CityGML_Core::_CityObject*. Thus, every *CityObject* makes a new association with two other classes, *EnergyDemand* and *WeatherData*. Therefore, attributes of a *VectorLayer* may include an integer identifier from the table *ng_cityobject* when they are put together into a *View*. Otherwise, this *id* column extracted from *ng_cityobject* will be *NULL*. In the case of *ThermalZone*, its layer would then have attributes from three tables, *ng_thermalzone*, *ng_cityobject* and *cityobject*. Secondly, and applicable only to one class, the *_AbstractBuilding* class in the Energy ADE inherits a CityGML class of the same name. This in turn implies that a layer constructed from this class will have information from up to four 3DCityDB tables, *ng_building*, *building*, *ng_cityobject* and *cityobject*.

For a KIT Profile feature whose layer is of type *VectorLayerNoGeom* or

DetailViewNoGeom, the spatial filter is foregone since neither layer type has geometry. Instead, gathering of attributes into a *View* hinges on the total number of records in the underlying 3DCityDB table. *VectorLayerNoGeom* layers will comprise attributes from the 3DCityDB relation intended to contain properties of the feature itself, and from *cityobject* and *ng_cityobject* for the same reason explained for a *VectorLayer*. *Facilities* is one example of a class that produces a *VectorLayerNoGeom*, it draws information from *ng_facilities*, *ng_cityobject* and *cityobject*. *DetailViewNoGeom* layers represent classes that are not a *CityObject* and have no spatial component. Strictly speaking, they should contain columns from only one table, but a few exceptions occur since the 3DCityDB encoding maps some classes fit for *DetailViewNoGeom* layers from *cityobject*. *EnergyDemand* is one case whose *DetailViewNoGeom* layers will extract attributes from *ng_energydemand* and *cityobject*, rather than from *ng_energydemand* only.

Figure 32 generalises the previous step for all four layer types. Depending on the presence of geometry, one of two pathways is chosen, but still leading to the same outcome. When SQL statements that generate a layer have been put together as described, triggers are then attached to each view by invoking the *generate_sql_triggers* function. Following this, metadata values corresponding to the columns specified in the previously prepared *INSERT* for the *layer_metadata* table are added to finalise the query. From the metadata presented in Section 3.2.3, it should however be noted that *DetailView* and *DetailViewNoGeom* layers do not have a feature type or top-level class. In addition, records for layer types which have no associated geometry do not have a *gv_name* in the layer metadata table. Also, non-spatial layers do not have values for *qml_symb* and *qml_3d*, which are QML files used for styling features in the QGIS map canvas. Ultimately, the collection of SQL code for creating a layer generated as detailed in preceding paragraphs composes one big SQL string which is subsequently executed by another function.

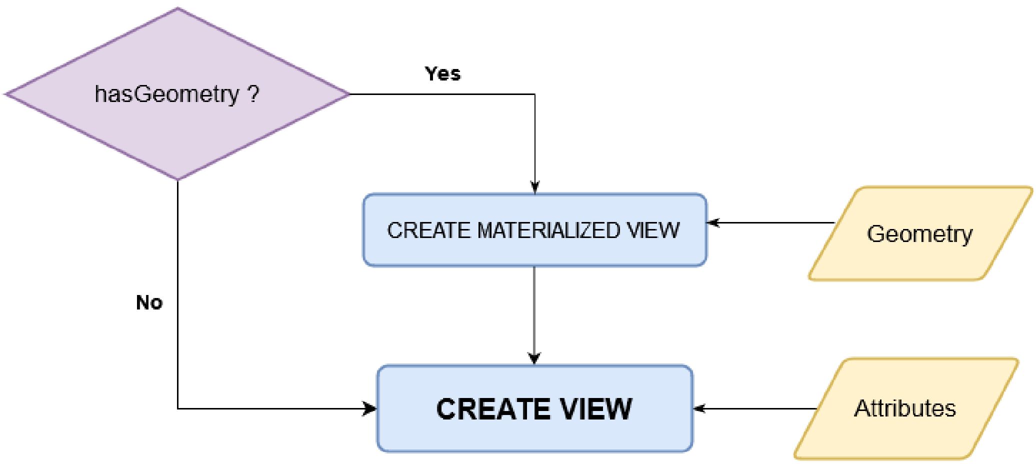


Figure 32: Workflow depicting the decision process behind creating a layer for the various layer types.

4.2.2.2 Layer Metadata

Metadata is created and inserted into the layer metadata table when a layer is created. This is the same relation discussed in Section 3.2.3 and used by the current version of 3DCityDB-Tools which does not support ADEs. Though the column for an ADE prefix is not used when recording metadata about CityGML layers, each entry in the table for an Energy ADE layer makes use of it. Instead of just *VectorLayer*, the other three layer types are introduced in the metadata table. For Energy ADE classes with no CityGML feature type or top-level class, the corresponding metadata columns are left empty. All features without at least one LoD defined by the data model are assigned *lodx*. This includes those without geometry, which in addition do not have any metadata pertaining to a geometry view, QML symbology form name and QML 3D symbology name. By extension, the layer metadata attribute which captures the latest refresh date of a *Materialized View* remains unused by layers with no geometry. The remaining columns of the layer metadata table are used as described in Section 3.2.3.

4.2.2.3 Updating a Layer

As alluded to in Sections 3.2.2 and 4.2.2.1, triggers are attached to a layer when it is created using the function *generate_sql_triggers*. When invoked, it iterates over the INSERT, UPDATE and DELETE commands and builds a SQL statement that creates a row-level trigger which in turn prescribes a trigger function to be subsequently executed for the each of the three data query operations. Triggers prevent a *View* from being updated directly and divert an INSERT, UPDATE OR DELETE query on any layer to the linked trigger function.

When an INSERT query is attempted on a layer to introduce new records into the view, the trigger for this operation is fired and in turn invokes the trigger function. Rather than completing the insertion, the trigger function raises an error which notifies the operator that QGIS Package does not permit this action as shown in Figure 33. On the other hand, delete queries on layers are allowed by `3DCityDB-Tools`, as are updates. When a query prompts deletion of rows from a layer, the associated trigger is set off and consequently it invokes the linked trigger function. Within the trigger function, the *id* of the record intended for deletion is passed on to a native 3DCityDB function named as *del_cityobject* and designed to erase records from the base relation. Taking table *ng_thermalboundary* as an example, this *id* is used in a call to the function *del_ng_thermalboundary* found in the underlying 3DCityDB schema, which then performs the operation. Figure 34 gives an illustration.

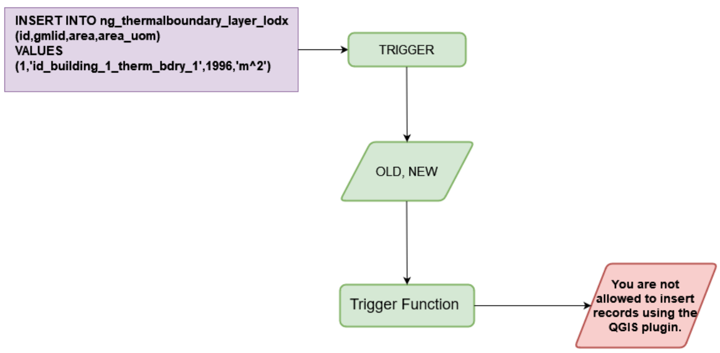


Figure 33: Workflow for thwarting an INSERT query on a layer.

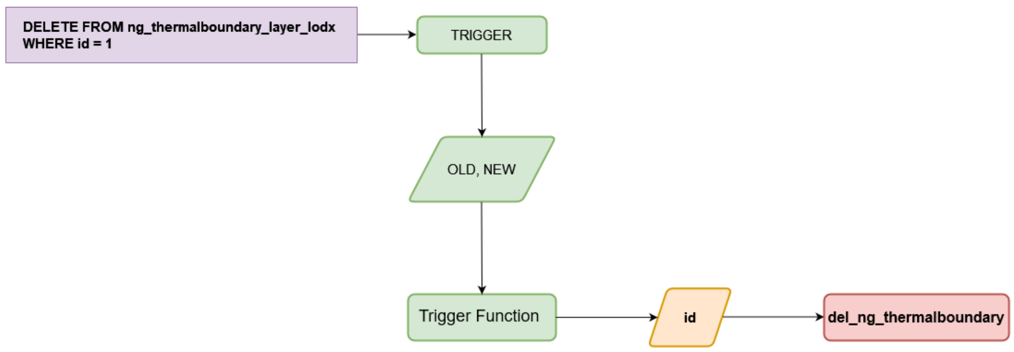


Figure 34: Workflow for committing a DELETE query to the underlying 3DCityDB tables of a layer.

For an UPDATE query, several functions simulate the effect of an updatable layer. To begin with, the query is broken down, separating attributes using a *Type* for each 3DCityDB table associated with the layer. This is done on the basis of a layer being a view constructed from at least two 3DCityDB tables, as is the case for many layers created for Energy ADE KIT Profile classes. All objects are then handed over to another function which in turn passes each *Type* to a function responsible for performing actual updates in a 3DCityDB table. Figure 35 illustrates how an UPDATE query is completed with the aid of an example. An UPDATE operation is performed to a layer for the *ThermalBoundary* class. A trigger stops the transaction and gives the trigger function access to the *OLD* and *NEW* records for the corresponding *id* in the query. Since a *ThermalBoundary* is a *CityObject*, the trigger function sieves through the query using two objects, one for each of the aforementioned classes. The objects are then given to a *View Update Function* whose purpose is to distribute the two objects to other functions that modify the 3DCityDB tables which provide attributes to the *View*. For this example, the tables are *cityobject* and *ng_thermalboundary*. The reader should also be made aware that the *CityObject* class in the Energy ADE has been left out of this scenario. This is because its 3DCityDB table only contains an *id* column, and 3DCityDB-Tools prohibits alteration of database keys.

Typically, each *Table Update Function* carries out a few checks before constructing and executing a SQL statement to alter information in a 3DCityDB table. For a layer constructed for an Energy ADE class containing at least one enumeration, any value to be inserted in the corresponding column through an update will be queried against an array of values specified by the data model. This prevents non-standardised values from being entered, thereby enforcing semantic interoperability. Furthermore, entries of quantities are only completed if a unit of measurement is included in the update statement. Otherwise, the function reports back to the user to provide the missing component. The object received by the function is also inspected to decide if any default values should be incorporated in the update statement that follows. Update functions *ThermalBoundary*, *SolidMaterial* and *ThermalZone* can serve as examples in explaining these three checks. Upon receiving a *ThermalBoundary* object with new information, *upd_t_ng_thermalboundary* inspects the *thermalboundarytype* property to assess if its value matches any of those in the *ThermalBoundaryType* enumeration. Similarly, *upd_t_ng_solidmaterial* ensures that for the properties *conductivity*, *density*, *permeance*, *specificheat* both a quantity and unit of measurement are provided. A default value of 1 indicating *true* is assigned to the *iscooled* and *isheated* attributes of table *ng_thermalzone* in *upd_t_ng_thermalzone* only when none has been specified in either case.

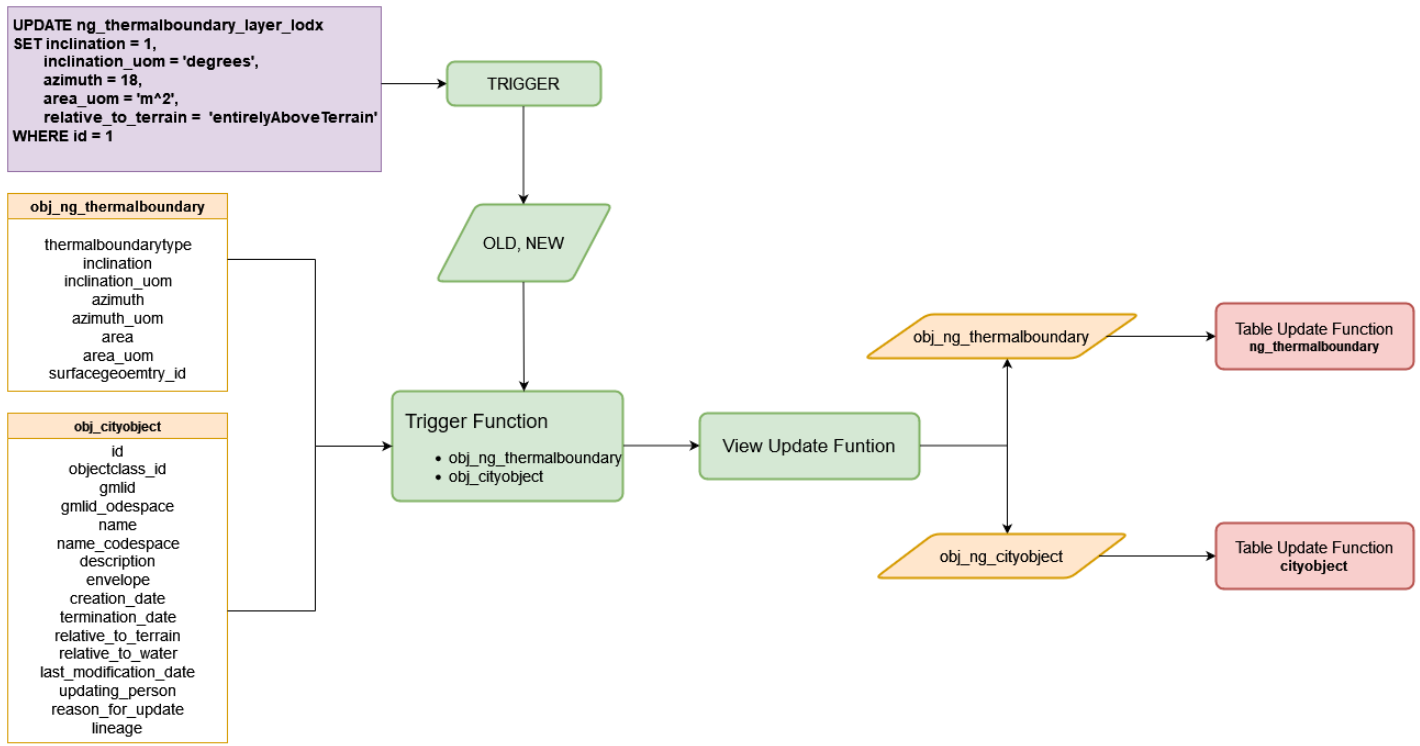


Figure 35: Workflow for committing an UPDATE query to the underlying 3DCityDB tables of a layer.

4.2.3 Front-End

4.2.3.1 QGIS Package Administrator

QGIS Package Administrator (Figure 22) lets users (un)install the back-end *qgis_pkg* schema in a PostgreSQL database in addition to user, schema and privilege management. Development of ADE support has brought more capabilities to QGIS Package without any visual modifications to this dialog. Previously, 24 scripts were run to create database object types, tables and views, functions and triggers for CityGML only. Now, 8 more files have been added to the same directory such that an ADE-enabled QGIS Package can be installed in the same way as before. An important addition to highlight is the *ade_feature_types* table. It stores every new feature type defined by an ADE installed in the database. This information is later used when layers are being created.

In the User Installation section of the dialog, a user schema can be created on the back-end as described by Agugiaro and Pantelios (2023) and in Section 3.3.1. It is the gateway through which non-administrative users take full advantage of QGIS Package. The ”Create schema” button (Figure 22) initiates events that lead to a new schema being created in the database for the specified user. Together with *layer_metadata*, numerous other relations which are *enum_config*, *enumeration_template*, *enumeration_value_template*, *codelist_lookup_config_template*, *codelist_template* and *codelist_value_template* are then sourced from *qgis_pkg* to the user schema. In addition, the views *v_enumeration_value* and *v_codelist_value* which are formed as joins between some of the above tables are created in the *usr_schema*. All these containers will also store metadata about ADE layers and values from lists in the KIT Profile data model when populated. Tables 9 to 14 show example entries in enumeration and codelist tables using *ThermalBoundaryTypeValue* and *EnergyCarrierTypeValue* respectively.

Table 9: Table enum_lookup_config example using ThermalBoundaryTypeValue

ade_prefix	source_class	source_table	source_column	target_table	key_column	filter_expression
ng	ThermalBoundary	ng_thermalboundary	thermalboundarytype	v_enumeration	value	data model = Energy ADE 1.0 AND name = ThermalBoundaryType

Table 10: Table enumeration_template example using ThermalBoundaryTypeValue

id	data_model	name	name_space
10	Energy ADE 1.0	ThermalBoundaryTypeValue	http://www.sig3d.org/citygml/2.0/energy/1.0/EnergyADE.xsd

Table 11: Table enumeration_value_template example using ThermalBoundaryTypeValue

id	code_id	value	description
53	10	interiorWall	Vertical partition separating two Thermal Zones of the same building

Table 12: Table codelist_lookup_config_template example using EnergyCarrierTypeValue

ade_prefix	source_class	source_table	source_column	target_table	key_column	filter_expression
ng	EnergyDemand	ng_energydemand	energycarriertype	v_enumeration	value	data model = Energy ADE 1.0 AND name = EnergyCarrierType

Table 13: Table codelist_template example using EnergyCarrierTypeValue

id	data_model	name	name_space
37	Energy ADE 1.0	EnergyCarrierTypeValue	NULL

Table 14: Table codelist_value_template example using EnergyCarrierTypeValue

id	code_id	value	description
1578	37	Electricity	NULL

4.2.3.2 Layer Loader

Layer loader, the dialog used to create, refresh and drop layers as well as import them into QGIS, has undergone minor visual modification in this study (Figure 36) while bringing the ability to interact with ADE-extended layers. Once a connection is made to the database and the user selects a 3DCityDB schema to load, the database is queried for the presence of any ADEs in the 3DCityDB relation *ade*. If at least one is found, the new section *ADE Selection* in the window is enabled so that its child element can later be accessed by the user to choose an ADE to load. The decision to place the *QGroupBox* in that position was largely informed by visual appeal. Regardless, the dialog was programmed in a way that adapts to the presence or absence of an ADE in the chosen 3DCityDB schema.

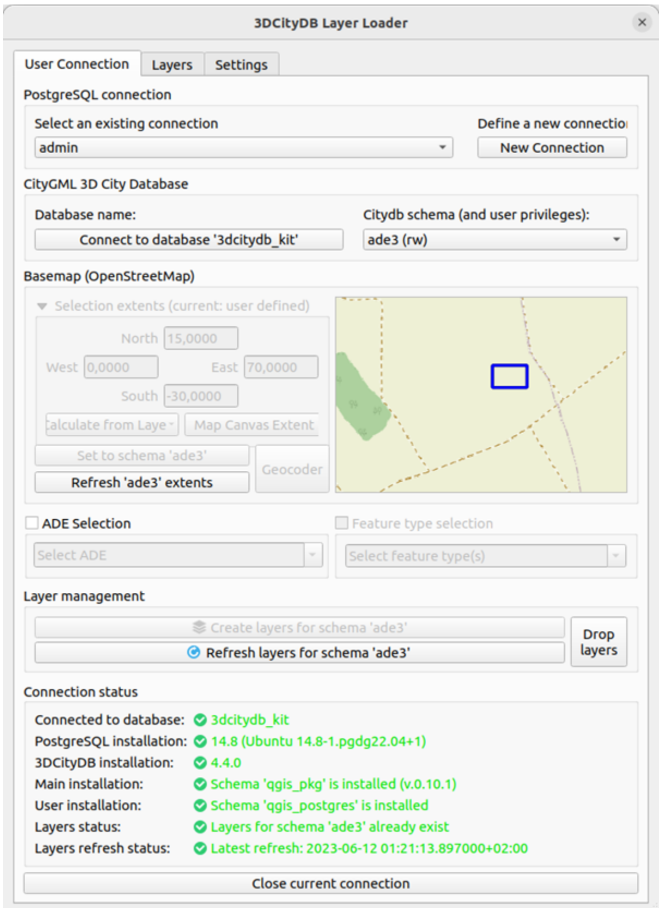


Figure 36: Modified User Connection tab in the Layer Loader dialog.

When a user chooses to select an ADE, *Feature type selection* is disabled and the *QgsCheckableComboBox* with default text *Select ADE* gathers all ADEs in the database to let the user choose one. As soon as an ADE is selected, *Feature type selection* is enabled. Only one ADE can be selected in a given procedure, a design decision justified in Section 4.1.5. To enforce this, the signal-slot mechanism of the Qt framework is leveraged. Selecting a second ADE emits a signal and the custom slot connected to it displays an information box which reports the above constraint and guides the user to have only one option checked. An alternative to the *QgsCheckableComboBox* could be a *QComboBox* that only allows one option to be chosen from a list. Nevertheless, its use has not been explored to provide a starting point for any further development of the plugin which attempts to support creating, refreshing and deleting layers for multiple ADEs concurrently. There will be no need to change any GUI elements, only what happens behind the interface.

After *ADE Selection* is enabled and an ADE chosen in the drop-down menu (Figure 37), the relation *ade_feature_types* is requested for all new feature types brought by that ADE. If any, *WeatherStation* in this case of the Energy ADE, they together with the CityGML feature types are placed in the *QgsCheckableComboBox* under *Feature type selection* (Figure 38). Layers for all feature types the user proceeds to select can then be created, refreshed, dropped or imported. Should there be no ADE found in the database, the *ADE Selection* remains disabled and *Feature type selection* will display only CityGML modules. After clicking a button to create, refresh or drop layers, the sequence of events that ensues depends on whether an ADE has been selected. Suppose there is no ADE, the plugin works just like its official release, as if no new functionality has been added. In reality, the concept of branching is applied in existing functions to determine whether to follow a path that produces ADE-enabled behaviour. Also, use of a separate thread and worker class as described in Section 3.3.2.1 is maintained for ADE-related tasks.

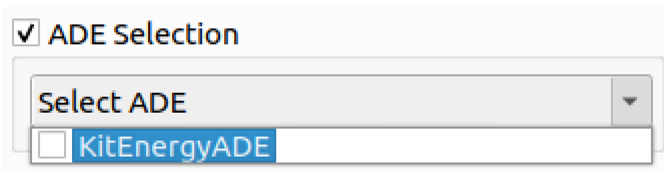


Figure 37: *QgsCheckableComboBox* which lists all available ADEs in the database.

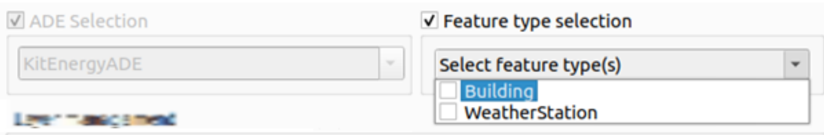


Figure 38: *QgsCheckableComboBox* which lists all available feature types in the database.

To create the 4 layer types, the existing function *evt_btnCreateLayers_clicked* (Listing 1) is programmed to follow a new branch to invoke another function that initiates the process in a separate thread. The *QThread* object is handed a *QObject* worker class, *CreateADELayersWorker*, whose *create_ade_layers_thread* method is connected to the *started* signal of the thread and invokes server-side functions that create the layers. The worker class method sends *SELECT* queries that call functions in the *qgis_pkg* schema to create layers for all selected feature types. This logic is similar to what is illustrated in Figure 24, except that here a decision is introduced after the button is clicked to determine whether to create a thread for CityGML or ADE layers.

```

def evt_btnCreateLayers_clicked(self) -> None:
    """Event that is called when the 'Create layers for schema {sch}' pushButton (btnCreateLayers) is pressed.
    """
    if self.gbxFeatSel.isChecked():
        # Update the FeatureTypeMetadata with the information about the selected ones
        tc_f.update_feature_type_registry_is_selected(self)
        selected_feat_types: list = gen_f.get_checkedItemsData(self.cbxFeatType)
        print('registry\n',self.FeatureTypesRegistry)

        if len(selected_feat_types) == 0:
            error_msg = f"You must select at least one Feature Type. Otherwise deactivate the Feature Type selection box."
            QMessageBox.warning(self, "User schema not found", error_msg)
            return None # Exit

    # Start the thread to create the layers (materialized views)
    if not(hasattr(self,'ADE_Registry')) or not(len(self.ADE_Registry.keys())):
        thr.run_create_layers_thread(self)
```

```
    else:
        thr.run_create_ade_layers_thread(self)

    return
```

Listing 1: Function invoked when the button to create layers in the GUI is clicked. It first determines if an ADE has been selected then proceeds to initiate an appropriate thread.

If no feature types are seleted, layers for all feature types available in the chosen 3DCityDB schema are created. Only two feature types are affiliated with the KIT Profile, *Building* and *WeatherStation*. For this reason, back-end functions invoked will produce layers whose feature type may only be either of the two. *ThermalZone* and *UsageZone* are respective examples of classes whose CityGML feature type is *Building* since they compose *_AbstractBuilding*, although for one layers of type *VectorLayer* are produced and *VectorLayerNoGeom* for the other. To create their layers as described in Section 4.2.2.1, the back-end function *create_layers_ng_building* is called.

An identical setup is in place for the client-side functions *evt_btnRefreshLayers_clicked* and *evt_btnDropLayers_clicked* to respectively refresh and delete layers in and from the user schema. They have an alternative branch that leads to a separate thread which is connected to a method in the worker class that takes care of refreshing or dropping layers. Listings 2 and 3 portray source code which applies the branching concept to refresh or drop layers.

```

def evt_btnRefreshLayers_clicked(self) -> None:
    """Event that is called when the 'Refresh layers for
   schema {sch}' pushButton (btnRefreshLayers) is pressed.
    """
    res = QMessageBox.question(self, "Layer refresh", "
   Refreshing layers can take long time.\nDo you want to
   proceed?")
    if res == 16384 and not(hasattr(self,'ADE_Registry')):
        thr.run_refresh_layers_thread(self)
    elif res == 16384 and (hasattr(self,'ADE_Registry')):
        thr.run_refresh_ade_layers_thread(self)

```

```
    return
```

Listing 2: Function invoked when the button to refresh layers in the GUI is clicked. It first determines if an ADE has been selected then proceeds to initiate an appropriate thread.

```

def evt_btnDropLayers_clicked(self) -> None:
    """
    Event that is called when the 'Drop layers for schema {
    sch}' pushButton (btnRefreshLayers) is pressed.
    """
    if not (hasattr(self, 'ADE_Registry')) or not (len(self.
    ADE_Registry.keys())):
        has_ade_layers_query = f'''
        SELECT COUNT(id) FROM
        {self.USR_SCHEMA}.layer_metadata
        WHERE cdb_schema = '{self.CDB_SCHEMA}'
        AND ade_prefix = '{'ng'}' AND gv_name != '{' '}' '''

        has_ade_layers = 0
        with self.conn.cursor() as cur:
            cur.execute(has_ade_layers_query)
            has_ade_layers = cur.fetchone()[0]

        if not(has_ade_layers):
            thr.run_drop_layers_thread(self)
        else:
            QMessageBox.information(self, 'Drop Layers',
            f'''Only layers affiliated with an ADE exist in
    user schema {self.USR_SCHEMA} for citydb schema {self.
    CDB_SCHEMA}. Select an appropriate ADE to drop the layers.
    ''')

    else:
        thr.run_drop_ade_layers_thread(self)

    return
```

Listing 3: Function invoked when the button to drop layers in the GUI is clicked. It first determines if an ADE has been selected then proceeds to initiate an appropriate thread.

Once layers are assembled and refreshed, the *Layers* tab of the same dialog is activated. A feature type with at least one existing layer and a level of detail can be selected as explained in Section 3.3.2.2. This includes *WeatherStation* and *lodx*. In addition to a *VectorLayer*, a

VectorLayerNoGeom for an Energy ADE class like *Facilities* can also be selected and imported into QGIS in this tab. Though there are four layer types for this ADE, each one is imported as an instance of the pyQGIS class *QgsVectorLayer*. It is given a data source, or a *QgsDataSourceUri* object, which establishes a channel to directly commit any layer updates to the database. The data source class is also used to indicate presence or absence of geometry in a layer using the *aGeometry* keyword parameter of its *setDataSource* method. This helps to distinguish layers with from those without geometry.

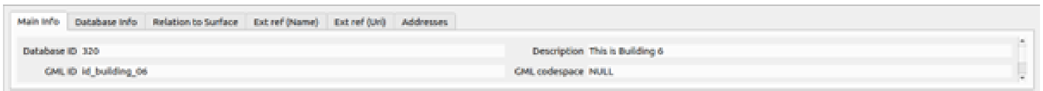
Furthermore, as layers are imported into QGIS, every *QgsVectorLayer* instance is linked to an *Attributes Form* whose name is extracted from *layer_metadata* and to any associated codelists and enumerations. To connect all layer types to their value lists, existing functions *create_layer_relation_to_codelists* and *create_layer_relation_to_enumerations* originally developed for a *VectorLayer* were reverse engineered where possible. In a few other cases, new functions were created and relations defined using the pyQGIS class *QgsRelation*. An example is given in Listing 4. The snippet of code shown is taken from the function *create_layer_relation_to_ng_heightaboveground* used to link two layers. One is an extended *Building* layer, and the other a *HeightAboveGround* layer. The *id* of a building is referenced by a corresponding foreign key, *building_heightabovegroun_id*. Doing so brings consistency with the data model which stipulates a one-to-many relationship between *_AbstractBuilding* and *HeightAboveGround*, similar to generic attributes in CityGML. In addition, connection these two layers makes *HeightAboveGround* attributes visible in the attribute form for a building as shown in Figure 39b.

```
rel = QgsRelation()
rel.setReferencedLayer(id=layer.id())
rel.setReferencingLayer(id=dv_layer.layerId())
rel.addFieldPair(referencingField='
    building_heightabovegroun_id', referencedField='id')
rel.setName(name='re_' + layer.name() + "_" + dv_layer.name()
    )
rel.setId(id="id_" + rel.name())
```

Listing 4: Use of QgsRelation to create a link between a HeightAboveGround object and a building.

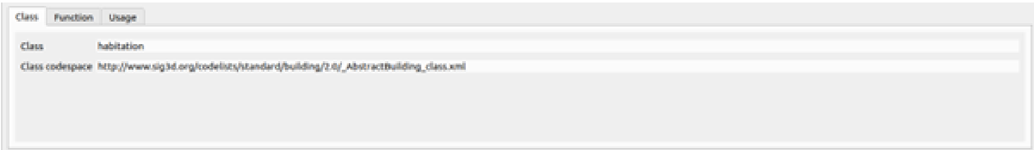
After layers are imported, they are organised in a layer tree using the same rules mentioned in Section 3.3.2.2. An illustration is given in Figure

11. Layer attributes can be accessed through a table or form like any other *QgsVectorLayer* object in QGIS. However, for each layer, a default form created by QGIS was modified to make a more user-friendly interface in which attributes are logically organised in tabs. Figures 39a to 39f show different sections of the form for the *_AbstractBuilding* class in the KIT Profile. In Figure 39a, the tabs contain attributes from the CityGML *Core* module. Figure 39b shows tabs for generic attributes associated with a *Building* and *BuildingPart*. Here, *FloorArea*, VolumeType and *HeightAboveGround* can be seen to have their own tabs as they are generic attributes as well. In fact, these two classes have their own layers which are *DetailViews* whose forms are also customised. For their corresponding tabs in the *_AbstractBuilding* form to be populated, a reference is created when layers are imported into QGIS. Figure 39d shows the remaining tab which displays attributes specific to an extended *_AbstractBuilding* class. Two attributes in particular are worth pointing out, *Building Type* and *Construction Weight*, as one has a codelist and the other an enumeration. Figures 39e and 39f show the drop-down menus for these two, which show descriptions rather than actual values.

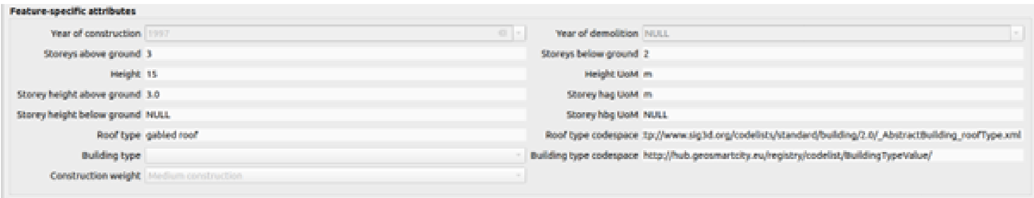


(a) Attributes inherited from *CityObject*.

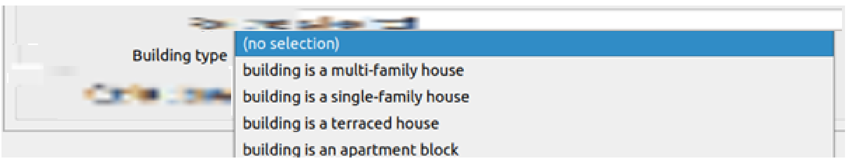
(b) Generic attributes of *_AbstractBuilding*.



(c) Class, Function and Usage attributes inherited from *_AbstractBuilding* in CityGML.



(d) Feature-specific attributes from both *_AbstractBuilding* in CityGML and in the KIT Profile.



(e) Options for the *Building Type* codelist.

(f) Options for the *Construction Weight* enumeration.

Figure 39: Customised Attributes Form for a layer constructed from the KIT Profile class *_AbstractBuilding*.

4.2.3.3 Bulk Deleter

To delete features in their entirety (geometry, topology, semantics and properties) from the database is the purpose of the Bulk Deleter, as is inherent in the name. Although not related to the Energy ADE, preliminary tests on how to structure this dialog (Figure 28) were part of this research to familiarize with the plugin. It should be noted that due to time constraints, this dialog has no ADE support. Since version 0.7, the Bulk Deleter is officially part of 3DCityDB-Tools.

By interacting with the map canvas manually, users can set a bounding box within which all contained features are to be deleted from a 3DCityDB instance. A geocoder with a dynamic window has been added also as part of this research to enhance the user experience by allowing users to quickly navigate to an area of interest in the case of city models spanning a large geographic extent. From text input in the Geocoder dialog (Figure 40), the OpenStreetMap (OSM) Nominatim API is queried for a matching place name or address using a *GET* request in which the search string is encoded. Matching results from OSM are sent through a spatial filter which uses the map canvas extent to discard irrelevant locations. Those that pass through the filter are then populated in a drop-down menu to allow the user to select the most appropriate one (Figures 41 and 42) and zoom to it on the map canvas.

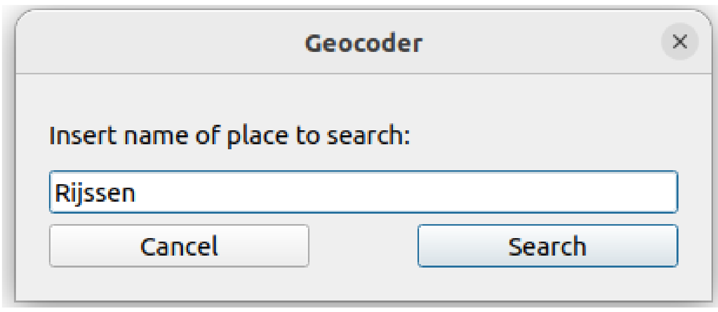


Figure 40: Geocoder dialog.

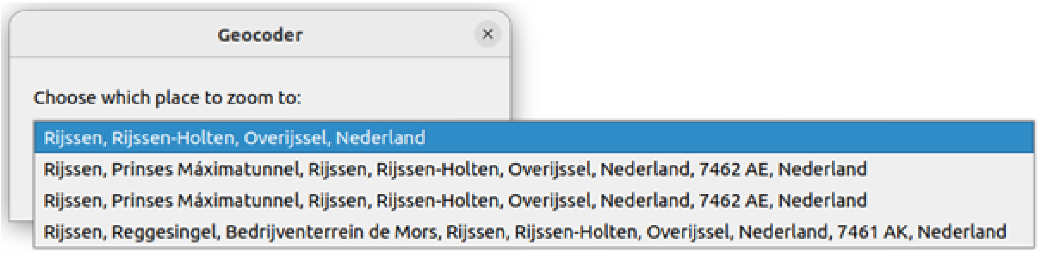


Figure 41: Geocoder dialog.

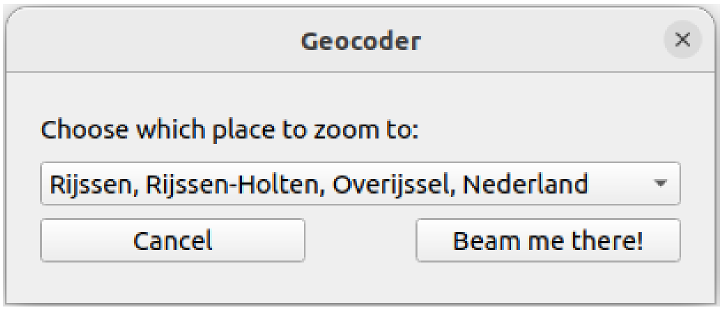


Figure 42: Geocoder dialog.

In the back-end, a native 3DCityDB delete function named as *del_cityobject* is invoked. It takes care of the hierarchies in CityGML by removing each feature in the bounding box from its respective table as well as all associated information in other tables. Furthermore, the option to instead truncate all tables in the database is at the disposal of the user from the dialog of the Bulk Deleter.

5 Conclusion

This research demonstrates the possibility of adding support for a CityGML Application Domain Extension to 3DCityDB-Tools for QGIS using the Energy ADE KIT Profile as a test case. Currently, the plug-in only implements functionalities for the CityGML data model, which restricts the degree to which users with limited technical knowledge of the 3DCityDB environment can further exploit extended CityGML datasets. Furthermore, a 3DCityDB instance extended by an ADE possesses an added dimension of complexity. Pantelios (2022) concludes that similar QGIS plug-ins, including the CityJSON Loader, have limited capabilities and user experience. In the context of ADE support, the same restraints exist. To address these issues for users, the following objectives were tackled in this research:

1. Conceptual definition of a strategy to add server-side support for an ADE to QGIS Package, the server-side component of 3DCityDB-Tools for QGIS.

   The concept of having a set of scripts which bring CityGML capabilities to QGIS Package was maintained for the Energy ADE KIT Profile. New scripts that embed extended functionality are also executed when the *qgis_pkg* schema is created on the server-side. In addition, a taxonomy for classifying various layer types for an ADE-enabled QGIS Package was established for the Energy ADE KIT Profile. Ultimately, this strategy allowed for the support of ADE layers by overcoming the different elements introduced by the ADE which include new *CityObjects* with and without geometry, extended *CityObjects*, as well as other classes that are not *CityObjects* but may have geometry. Lastly, users would also be able to update layer attributes using QGIS Package.

2. Develop an ADE-enabled QGIS-Package, with focus on the Energy ADE KIT Profile.

   Object types, tables, triggers and functions which bring ADE functionalities were integrated into QGIS Package. They enable creating, refreshing or deleting layers. Furthermore, INSERT, DELETE and UPDATE queries on layers, which are actually views containing information from multiple 3DCityDB tables, are handled by the added functionalities to aid layer management. To complement these tasks, metadata for ADE layers is also managed.

3. Conceptual definition of a strategy to add client-side support for an ADE to the front-end of the 3DCityDB-Tools for QGIS plugin.

   The strategy was for users to be able to create, refresh or drop ADE layers from within QGIS, as well as import them. In addition, modifying attributes of or deleting existing features in a layer would be facilitated.

4. Develop an ADE-enabled 3DCityDB-Tools for QGIS front-end.

   Two new GUI elements are introduced to the Layer Loader dialog. In a given session, these elements help functions involved in creating, refreshing or deleting layers to keep track of the selection of an ADE and by extension to determine sequence of events. Behaviour of the Layer Loader in creating, refreshing or dropping layers is governed by branching to select a sequence of events in the absence or presence of an ADE. QGIS forms for feature attribute editing were customised to enhance the user experience but also to enforce constraints that match those of the database tables connected to a layer.

5. Contribute to further testing and extending and improving existing functionalities.

   This research contributed to tests around how to structure GUI elements in the Bulk Deleter dialog and connect them to front-end and back-end functionality. The geocoding feature was first investigated in these tests as well. Both are officially part of 3DCityDB-Tools for QGIS since version 0.7.

On the basis of these outcomes, 3DCityDB-Tools can support ADEs to a greater extent. Nevertheless, there are a few considerations to emphasise from this research.

This study focuses on a subset of the Energy ADE, the KIT Profile, which has significantly fewer classes, associations, enumerations and codelists. As an example, 3 feature types in the Energy ADE, *EnergySystem*, *EnergyConversionSystem* and *EnergyDistributionStorageSystem* which together define 32 classes are completely left out of the KIT Profile. Also, extending the 3DCityDB with the full Energy ADE creates 79 new tables, in contrast to only 33 for the KIT Profile. These disparities raise uncertainty over applicability of the methodology and implementation in this study to the Energy ADE. Furthermore, investigating the possibility of ADE support using just one ADE does not necessarily yield an outcome which is representative of a real-world scenario that puts semantic 3DCM to use in multiple disciplines. This prompts more questions to be asked with regards to a generic approach for ADE support in 3DCityDB-Tools.

Other considerations relate to the back-end architecture of the plugin. Figure 35 shows trigger, and view and table update functions used to make layers updatable. The trigger function sends objects to the view update function which in turn distributes them to table update functions. While this setup has been shown to work, it has a component that may be redundant. The only task of the view update function is to act as a conduit between a trigger function and one or more table update functions. Using 3 functions to execute updates for ADE layers is an idea borrowed from what was the current version of QGIS Package in the early phases of this study. By cutting out the view update function, table update functions can be connected to trigger functions and receive objects directly from there. This could reduce the amount of code required in future to add support for more ADEs in QGIS Package.

Remaining points to note are to do with the client-side. Concerning the design decision to restrict users to have layers for either the CityGML base profile or an ADE, further reasoning may be required in the presence of two or more ADEs. The metadata table in the user schema on the back-end is populated when layers for CityGML or an ADE are created on the front-end. By extension, the table can only have metadata about layers for one data model. If a second ADE is introduced, it is unknown whether the above design decision will require rethinking. Should users be allowed to create layers for more than one ADE simultaneously? Should the layer metadata table accommodate CityGML and multiple ADEs at the same time? What are the implications for other processes carried out within 3DCity-Tools?

Also, for semantic city models spanning a large geographic extent, refreshing layers or using the Bulk Deleter to remove features from a 3DCityDB may be time-consuming based on present experiences. Furthermore, the order of the Layer Loader tabs affects the User Experience (UX) to an extent. In the Settings tab, users can choose a data model for which to load codelists. However, layers are imported in the previous tab. Codelist values for an ADE will not appear in an *Attributes Form* if the data model is not selected before importing layers into QGIS. The user has to repeat the process so as to load the data model first. In addition, only one data model can be selected at a time. How then can a user access both CityGML and ADE codelists when the 3DCityDB is extended?

Moreover, the landscape around the plugin is evolving. 3DCityDB-Tools is currently based on versions 2.0 and 4.4 of CityGML and the 3DCityDB respectively. However, CityGML 3.0 has taken over as the latest release of the standard, and 3DCityDB 5.0 is under development. On one hand, CityGML 3.0 revises numerous existing modules including *Building* and introduces four new ones. For example, some classes like *Occupancy* and *HeightReferenceValue* currently in the Energy ADE are now incorporated in a new CityGML module called *Construction*. The latest version of CityGML also presents new *Space* and *Geometry* concepts. Generally, a space is a class associated with a geometry, and *CityObjects* like *_AbstractBuilding* inherit a *Space*. This setup diverts from what is seen in CityGML 2.0, where features are directly associated to geometric primitives. Regarding extensibility, CityGML 3.0 specifies ports where ADEs can dock onto more explicitly.

On the other, a revamped 3DCityDB for CityGML 3.0 contains four main tables, which is a drastic reduction in complexity comparing it with the current version 4.x. One table for all features and objects, another for all attributes and associations, a third for all geometries and the remaining one for colours and textures. Figure 43 gives a visual contrast between the current and upcoming 3DCityDB versions. Nagel and Zhihang (2023) highlight that while this setup is simple, efficient and easier-to-use with default GIS tools, substantially more work is required to adapt these tools to the redesigned 3DCityDB schema. These two developments, 3DCityDB 5.0 and CityGML 3.0, call into question the architecture of 3DCityDB-Tools and the outcome of this study with respect to future work.

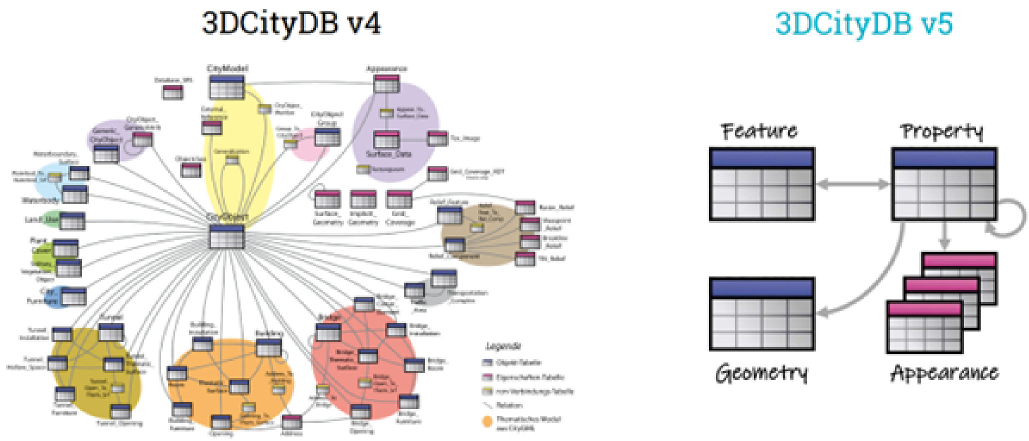


Figure 43: Current versus upcoming versions of the 3DCityDB. Source: (Nagel & Zhihang, 2023)

5.1 Future Work

By analysing the above, suggestions can be put forward for future work. A starting point may be to implement support for the whole Energy ADE, plus another one, to gauge scalability with respect to generic ADE support in the plugin. UtilityNetwork can serve as a good reference for a second ADE as it has been well-studied in literature (Boates et al., 2018; Den Duijn et al., 2018; Kutzner et al., 2018).

To reduce amount of code on the back-end, one solution could be to thoroughly analyse that sub-system to identify where operations can be made more elegant. Alternatives may involve exploring a different setup on the server-side, or consideration of a more compact SQL encoding of CityGML with fewer tables. Both options could, however, lead to a complete overhaul of 3DCityDB-Tools. Whether this is feasible or not might also be worth evaluating, especially considering the number of 3DCityDB tables introduced by the full Energy ADE.

A different server-side setup that could be investigated is having all geometries of a feature in one layer. This implies that only one *Materialized View* and layer are created for each feature, but the option for a user to choose one LoD to load on the client-side would be retained. CityGML 3.0 and 3DCityDB 5.0 could be used to explore the feasibility of the other suggested development pathway.

On the client-side, several improvements can be explored. First is the use of concurrency for tasks such as refreshing layers and deleting features from the 3DCityDB in bulk. Performance gains from this could be beneficial to users and developers. Modification of the GUI might also be investigated. A dynamic Layer Loader interface, similar to what is described in Section 4.2.3.3 for the Geocoder, could enhance the UX. One issue that would be addressed by a dynamic GUI is the placement of the Layers and Settings tabs. Database connection elements and settings, which might include choosing an ADE to load, can be first thing a user sees in this dialog. In addition, this may also allow only one map canvas to be used to create and import layers instead of the two canvases in the User Connection and Layers tabs.

A Reproducibility self-assessment

A.1 Marks for each of the criteria

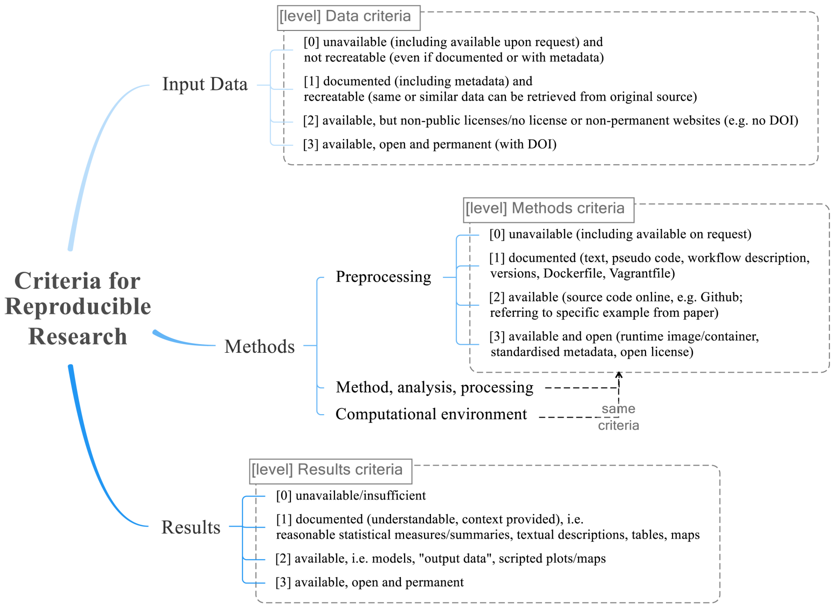


Figure 44: Reproducibility criteria to be assessed.

Grade/evaluate yourself for the 5 criteria (giving 0/1/2/3 for each):

2/3 input data

1/3 preprocessing

3/3 methods

3/3 computational environment

3/3 results

A.2 Self-reflection

A.2.1 Input Data

Data used in this research was provided by a supervisor of this research. Though it is not publicly stored online, it can be made available on request.

A.2.2 Methods

A part of this research that qualifies as preprocessing is requirements gathering. It was conducted in meetings and discussions, thus the information is poorly documented and not available.

With respect to the methods implemented in this research, a dedicated GitHub repository versioning and storing source code. The software developed in this research remains open source, and as such can be freely redistributed and/or customised within the terms stipulated by version 2 of the GNU General Public License as published by the Free Software Foundation.

Relating to the computational environment, all development was based on open source software.

A.2.3 Results

All modifications to the 3DCityDB-Tools plugin together with this report can be considered the final result. The plugin and the document are publicly available and accessible on GitHub and the Delft University of Technology repositories.

References

Agugiaro, G., Benner, J., Cipriano, P., & Nouvel, R. (2018). The Energy Application Domain Extension for CityGML: enhancing interoperability for urban energy simulations. *Open Geospatial Data, Software and Standards*, *3*(1), 2. https://doi.org/10.1186/s40965-018-0042-y

Agugiaro, G., & Pantelios, K. (2023). Quick installation and user guide. https://github.com/tudelft3d/3DCityDB-Tools-for-QGIS/blob/master/user_guide/3DCityDB-Tools_UserGuide_0.8.1.pdf

Ahmad, T., & Zhang, D. (2020). A critical review of comparative global historical energy consumption and future demand: The story told so far. *Energy Reports*, *6*, 1973–1991. https://doi.org/https://doi.org/10.1016/j.egyr.2020.07.020

Benner, J. (2018). CityGML Energy ADE V. 1.0 Specification.

Biljecki, F., Kumar, K., & Nagel, C. (2018). CityGML Application Domain Extension (ADE): overview of developments. *Open Geospatial Data, Software and Standards*, *3*(1), 13. https://doi.org/10.1186/s40965-018-0055-6

Biljecki, F., Stoter, J., Ledoux, H., Zlatanova, S., & Çöltekin, A. (2015). Applications of 3D City Models: State of the Art Review. *ISPRS International Journal of Geo-Information*, *4*(4), 2842–2889.

Boates, I., Agugiaro, G., & Nichersu, A. (2018). Network modelling and semantic 3d city models: Testing the maturity of the utility network ade for citygml with a water network test case. *ISPRS annals of photogrammetry, remote sensing & spatial information sciences*, *4*(4).

Catita, C., Redweik, P., Pereira, J., & Brito, M. C. (2014). Extending solar potential analysis in buildings to vertical facades. *Computers & Geosciences*, *66*, 1–12.

Den Duijn, X., Agugiaro, G., & Zlatanova, S. (2018). Modelling below- and above-ground utility network features with the citygml utility network ade: Experiences from rotterdam. *Proceedings of the 3rd International Conference on Smart Data and Smart Cities, Delft, The Netherlands*, *43*, 50.

Geiger, A., Benner, J., Häfele, K.-H., & Hagenmeyer, V. (2018). Thermal energy simulation of buildings based on the city-gml energy application domain extension. *BauSIM2018–7. Deutsch-Österreichische IBPSA-Konferenz: Tagungsband. Hrsg.: P. Von Both*, 295–302.

González-Torres, M., Pérez-Lombard, L., Coronel, J. F., Maestre, I. R., & Yan, D. (2022). A review on buildings energy information: Trends, end-uses, fuels and drivers. *Energy Reports*, *8*, 626–637. https://doi.org/https://doi.org/10.1016/j.egyr.2021.11.280

Graham, D. (1989). Incremental development: Review of nonmonolithic life-cycle development models. *Information and Software Technology*, *31*(1), 7–20.

Gröger, G., Kolbe, T. H., Nagel, C., & Häfele, K.-H. (2012). OpenGIS City Geography Markup Language (CityGML) Encoding Standard, Version 2.0.0. *OGC Document No. 12-019*, 344. https://portal.opengeospatial.org/files/?artifact_id=47842

Huld, T., Müller, R., & Gambardella, A. (2012). A new solar radiation database for estimating pv performance in europe and africa. *Solar Energy*, *86*(6), 1803–1815. https://doi.org/https://doi.org/10.1016/j.solener.2012.03.006

Jakubiec, J. A., & Reinhart, C. F. (2013). A method for predicting city-wide electricity gains from photovoltaic panels based on lidar and gis data combined with hourly daysim simulations. *Solar Energy*, *93*, 127–143. https://doi.org/https://doi.org/10.1016/j.solener.2013.03.022

Kim, D. W., Kim, Y. M., & Lee, S. E. (2019). Development of an energy benchmarking database based on cost-effective energy performance indicators: Case study on public buildings in south korea. *Energy and Buildings*, *191*, 104–116. https://doi.org/https://doi.org/10.1016/j.enbuild.2019.03.009

Kolbe, T. H. (2009). Representing and exchanging 3d city models with citygml. In *3d geo-information sciences* (pp. 15–31). Springer.

Korfiati, A., Gkonos, C., Veronesi, F., Gaki, A., Grassi, S., Schenkel, R., Volkwein, S., Raubal, M., & Hurni, L. (2016). Estimation of the global solar energy potential and photovoltaic cost with the use of open data. *International Journal of Sustainable Energy Planning and Management*, *9*, 17–30.

Kutzner, T., Hijazi, I., & Kolbe, T. H. (2018). Semantic modelling of 3d multi-utility networks for urban analyses and simulations: The citygml utility network ade. *International journal of 3-D information modeling (IJ3DIM)*, *7*(2), 1–34.

Ledoux, H., Arroyo Ohori, K., Kumar, K., Dukai, B., Labetski, A., & Vitalis, S. (2019). Cityjson: A compact and easy-to-use encoding of the citygml data model. *Open Geospatial Data, Software and Standards*, *4*(1), 1–12.

Machete, R., Falcão, A. P., Gomes, M. G., & Rodrigues, A. M. (2018). The use of 3d gis to analyse the influence of urban context on buildings' solar energy potential. *Energy and Buildings*, *177*, 290–302.

Mainzer, K., Fath, K., McKenna, R., Stengel, J., Fichtner, W., & Schultmann, F. (2014). A high-resolution determination of the technical potential for residential-roof-mounted photovoltaic systems in germany. *Solar Energy*, *105*, 715–731.

Martıénez-Rubio, A., Sanz-Adan, F., Santamarıéa-Peña, J., & Martıénez, A. (2016). Evaluating solar irradiance over facades in high building cities, based on lidar technology. *Applied energy*, *183*, 133–147.

Mavromatidis, G., Orehounig, K., Richner, P., & Carmeliet, J. (2016). A strategy for reducing co2 emissions from buildings with the kaya identity – a swiss energy system analysis and a case study. *Energy Policy*, *88*, 343–354. https://doi.org/https://doi.org/10.1016/j.enpol.2015.10.037

Nagel, C., & Zhihang, Y. (2023). *3dcitydb 5.0 workshop* [5CC+].

Nguyen, H. T., & Pearce, J. M. (2012). Incorporating shading losses in solar photovoltaic potential assessment at the municipal scale. *Solar Energy*, *86*(5), 1245–1260.

OECD. (2020). Cities in the world: A new perspective on urbanisation. https://doi.org/https://doi.org/10.1787/d0efcbda-en

Open Geospatial Consortium. (1999). Opengis simple features specification for sql. revision 1.1.

Pantelios, K. (2022). Development of a QGIS plugin for the CityGML 3D City Database. http://resolver.tudelft.nl/uuid:fb532bef-81b9-482b-921a-e7ce907cb544

QGIS. (2023). *Qgis python plugins repository*. Retrieved May 18, 2023, from https://plugins.qgis.org/plugins/

QGIS-Python-API. (2018). *Class: Qgsvectorlayer*. Retrieved April 20, 2023, from https://www.qgis.org/pyqgis/3.0/core/Vector/QgsVectorLayer.html

Qt-Project. (n.d.). *Qt project documentation*. Retrieved April 20, 2023, from https://doc.qt.io/qt-6/qmlapplications.html

Redweik, P., Catita, C., & Brito, M. (2013). Solar energy potential on roofs and facades in an urban landscape. *Solar energy*, *97*, 332–341.

Sherman, G., Sutton, T., & BLAZEK Rand LUTHMAN, L. (2005). Quantum gis user guide–version 0.7. 4 seamus.

Stadler, A., Nagel, C., König, G., & Kolbe, T. H. (2009). Making interoperability persistent: A 3d geo database based on citygml. In *3d geo-information sciences* (pp. 175–192). Springer.

The 3D City Database. (n.d.). *The 3d city database user manual*. Retrieved May 18, 2023, from https://3dcitydb-docs.readthedocs.io/en/latest/index.html

The PostgreSQL Global Development Group. (2023a). *Create type*. Retrieved June 2, 2023, from https://www.postgresql.org/docs/current/sql-createtype.html

The PostgreSQL Global Development Group. (2023b). *Create view*. Retrieved June 2, 2023, from https://www.postgresql.org/docs/current/sql-createview.html

The Qt Company. (2023a). *QPushButton Class*. Retrieved May 19, 2023, from https://doc.qt.io/qt-6/qpushbutton.html

The Qt Company. (2023b). *Signals Slots*. Retrieved March 13, 2021, from https://doc.qt.io/qt-5/qgradient.html

van Loenen, B. (2006). *Developing geographic information infrastructures: The role of information policies*. IOS Press.

Vitalis, S., Arroyo Ohori, K., & Stoter, J. (2020). Cityjson in qgis: Development of an open-source plugin. *Transactions in GIS*, *24*(5), 1147–1164.

Welle Donker, F. (2018). Funding Open Data. https://doi.org/10.1007/978-94-6265-261-3_4

Widl, E., Agugiaro, G., & Peters-Anders, J. (2021). Linking semantic 3d city models with domain-specific simulation tools for the planning and validation of energy applications at district level. *Sustainability*, *13*(16), 8782.

Willman, J. M. (2022). *Beginning PyQt*. Apress. https://doi.org/10.1007/978-1-4842-7999-1

Yao, Z., Nagel, C., Kunde, F., Hudra, G., Willkomm, P., Donaubauer, A., Adolphi, T., & Kolbe, T. H. (2018). 3DCityDB - a 3D geodatabase solution for the management, analysis, and visualization of semantic 3D city models based on CityGML. *Open Geospatial Data, Software and Standards*, *3*(1), 5. https://doi.org/10.1186/s40965-018-0046-7

Yuan, J., Farnham, C., Emura, K., & Lu, S. (2016). A method to estimate the potential of rooftop photovoltaic power generation for a region. *Urban Climate*, *17*, 1–19.

Zanzoterra, S. (2018). Evaluation of qt as gui framework for accelerator controls.

Zhang, X. Q. (2016). The trends, promises and challenges of urbanisation in the world. *Habitat international*, *54*, 241–252.