Università
della
Svizzera
italiana

**Software
Institute**

SOFTWARE & DATA CITIES

Visualizing Information Systems as Evolving 3D Cities

Susanna Ardigó

June 2021

Supervised by
Prof. Dr. Michele Lanza

Co-Supervised by
Dr. Csaba Nagy

SOFTWARE & DATA ENGINEERING MASTER THESIS

Abstract

Software systems are intangible as they are composed only of code, files and folders. They are also often large and complex. Modern system are built around data of different types, saved inside the repository and in separate databases. Software visualization can help in understanding their structure and identifying where they need improvements.

In the previous decades, the idea of the city metaphor has emerged. It has been primarily used to show the evolution of a software system, and some experimental approaches applied it to databases too. However, in modern information systems, these two essential parts coexist and interact with each other. Current visualizations are limited to only one single component and lack in showing the relations between them. The data stored inside the repository is also not considered in the analysis.

The goal of this master thesis is to visualize the evolution of software systems and their data, stored inside the repository and in databases, as cities. The relations of the two components are visualized, showing how the two parts interact with each other.

This new non-trivial augmented metaphor gives the visualization a new layout that allows the users to understand both software, data and how they grow and interact with each other.

Dedicated to my family that has always
believed in me and allowed me to pursue
my dreams

Acknowledgements

This historical period, characterized by the coronavirus COVID-19 pandemic, is not optimal for working on a master thesis. Thanks to these incredible people, it was not an issue.

I want to start by thanking Prof. Dr. Michele Lanza for being a source of inspiration and motivation since the very first Atelier lecture I attended on the first day of my Bachelor. Your design course convinced me to enroll in the Master of Software & Data Engineering when I was confused about the choice. Thank you for being my supervisor and allowing me to work on this fantastic project. I appreciate your availability, constructive feedback, and motivation to push beyond the current limits and constantly improve. During this thesis, I have grown personally and academically. I hope I will work again with you in the future.

I want to thank Dr. Csaba Nagy for being my co-supervisor. You have been a valuable support for the whole path of my master thesis. I really appreciate every time you were there, ready to help me with anything I could need, wearing a smile and armed with patience.

Thank you to the Professors of the Faculty of Informatics, your passion for teaching has helped me fall in love with this subject. You have created the perfect environment for learning and discussing with colleagues.

A special thanks to the Dean's Office. You have helped me to go through University and living in Switzerland. Your support has been vital during these years. You are the students' guardian angels, constantly checking on us, available to listen, give us suggestions and support. You are the best team anyone could ever wish for.

Thank you to my parents, Nadia e Walter, for making me what I am today. I was raised seeing you working and studying. It made me dream about spending my life studying and showed me that knowledge is never enough. You are smart, intelligent, and able to solve any problem. Thank you for allowing me to study in a different country. You supported me through all these years. I know you still struggle to understand what I do, but I am glad I studied a field different than the family's field. Thank you to my sister Francesca for supporting me and helping with anything I needed. We are opposites, but that's our strength.

Last, thanks to my friends for the good times spent during these years. Your support was significant and very much appreciated.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Modern software systems are big, complex, and constantly evolving, which makes them hard to maintain. During the lifetime of a system, many developers work on it, sometimes simultaneously, on different parts that interact with each other. Team leaders need to supervise how developers maintain and develop these parts. Being able to visualize them can aid in understanding their structure, how they interact with each other, and where they need improvements.

1.1 Software & Data

Source code is human-readable text that can be structured in many different ways depending on the language. Various programming languages can coexist in the same application. Software systems are composed of files saved in nested folders. Software engineering aims at designing and engineering the birth, growth, and maintainability of systems.

Data can be stored in many different forms depending on the structure needed. A simple text file with information can be considered as a form of a database. This method is widely used at the birth of systems when they are still under construction and do not yet have real data. However, files that store data are part of the repository, making modifications easy for developers.

With the growth of the systems, developers then introduce proper databases. In the past decades, developers mostly used relational databases. In a relational database, data is stored in tables. A table represents a type of data entity. Each row of the table represents an instance of that type of entity, and columns are their attributes. SQL provides syntax to create structured queries for retrieval, merging, and filtering data from tables. This type of database is typically visualized in a 2D form with ER diagrams where each table is represented by the columns as an entity box and the relations are shown by lines that link the columns of different tables.

New NoSQL technologies have been introduced, which allow databases to store data in different forms, for example, graphs, documents, or objects. These technologies do not need schemas that allow unstructured and partially structured data to be efficiently stored, queried, and retrieved.

1.2 Visualization

Software is visualized differently depending on the purpose. The most basic 2D visualization of object-oriented programs is the class diagram, an efficient representation of the architecture. The size of the system is directly proportional to the size of the diagram itself. A limitation of this visualization is that it shows too much detail. To visualize more concepts, more abstraction is needed to reduce the overall complexity. We can achieve this by extracting metrics from the source code to represent the content of a file with a numeric value. Examples of metrics are the lines of code, number of methods, instance variables, and loops.

Software systems are constantly evolving, making evolution an important aspect that needs to be shown in order to understand their lifetime. An example is the evolution matrix [29] where each class

is represented in a box whose height and width are determined by metrics. These boxes are placed in a matrix with the classes as rows and versions as columns.

Databases are visualized for various purposes. An ER diagram is the most common visual representation and is used to understand the schema. There are approaches to visualize the evolution, such as Dahlia [33, 34], that show two different cities of a database and an application that do not share any territory. Selecting a class building highlights the tables that the class interacts with and shows information about the queries and their location in the code.

1.2.1 City Metaphor

Cities are complex evolving systems made of blocks of buildings of many different sizes that are constructed, changed, maintained, and destroyed. Systems can also be visualized through the metaphor of evolving cities. This idea fits best in a 3D visualization because the addition of the third dimension allows more information to be displayed in a single frame. With the help of this metaphor, it is easy to see the structure and the evolution of a system.

This representation has been used for both software systems [41, 53] and databases [33]. An example is CODECITY [53], shown in Figure 1.1, a 3D city of the software system.

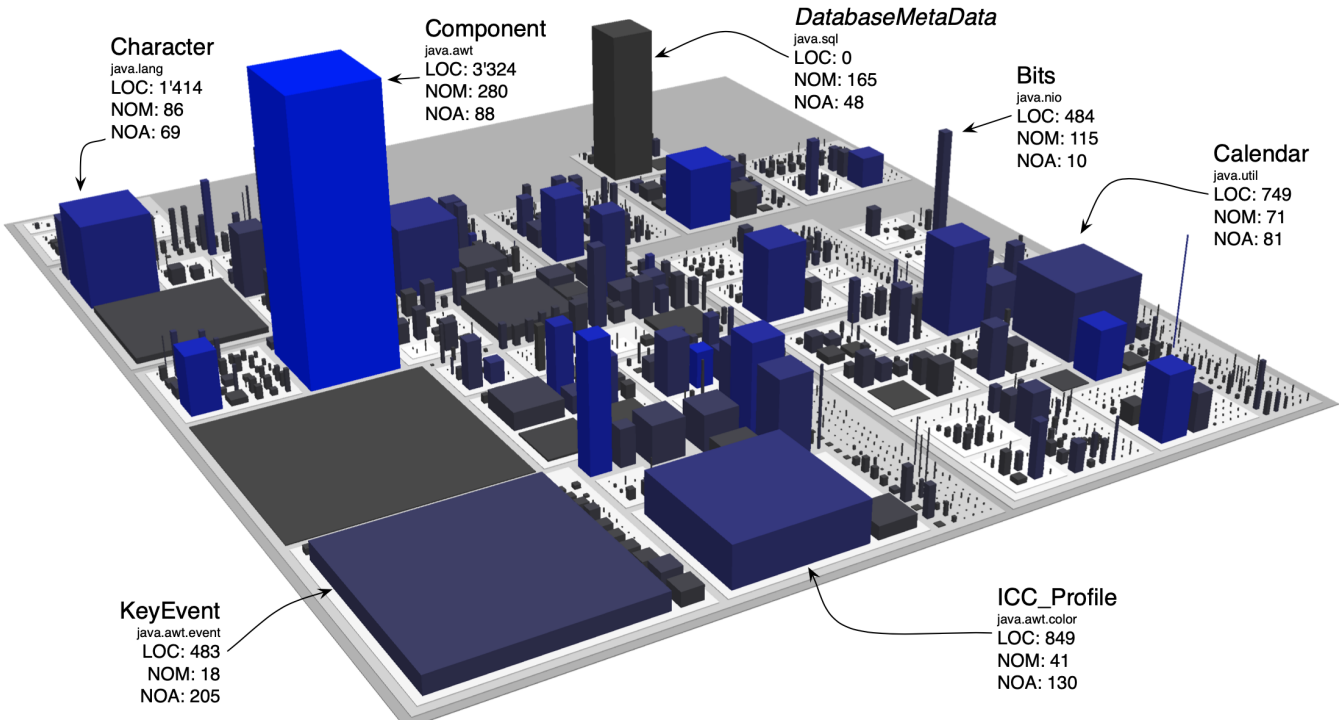


FIGURE 1.1: CODECITY [53]

1.3 Goal of this Thesis

Modern systems are built around data, making databases central components of modern applications. Nowadays, software and data coexist in systems and interact with each other frequently. They grow together and complement each other. Software systems store information inside the repository, in data files, and outside, in databases. This is not shown in the current visualizations; hence they are limited to showing either the source code of the application or the database.

The main goal of this thesis is to visualize evolving cities where software and data are both represented in different forms to understand how these two separate components are created and developed. This allows a deeper understanding of how data fuels software systems, where it is stored, and which part of the code interacts with it.

To achieve this goal, first, we need to model different types of data sources (for example, files and databases), and their interactions with the source code. We need different approaches for several database technologies available. Second, we extract relevant metrics for describing the data and investigate various visualization techniques. The representation needs to be intuitive, clean, and efficient.

1.4 Structure of the Document

In Chapter 2, we describe the state of the art in the visualization of software systems, databases, and the techniques used to visualize their interactions.
In Chapter 3, we describe our approach and the tool developed.
In Chapter 4, we demonstrate our approach by analyzing open-source software systems and our tool.
In Chapter 5, we present the conclusion of the thesis and possible directions for future work.

Chapter 2

State of the Art

The visualization of software systems and databases are two separate fields that focus mainly on different aspects that share the primary goal of visualization. In Section 2.1, we discuss the history of visualization of software systems, while in Section 2.2, we discuss the history of visualization of databases.

2.1 Software Visualization

Software systems visualization was born in the 1950s in the form of 2D diagrams. The first approach represented the programs with flowcharts. The first known program representation belongs to Haibt, shown in Figure 2.1. Each part of the code sits in a rectangle that points to the next part of the code. It gives developers the possibility to understand the behavior of the source code from the starting point to the ending [20]. Knuth used a similar idea, creating a tool that generated the visualization, like flowcharts, of software documentation [26].

Nassi and Schneiderman, a decade later, presented a tool to visualize the program execution shaped like a rectangle [37]. This rectangle is divided into sub-rectangles that represent each part of the code. The execution starts at the top, with a sub-rectangle, and ends at the bottom, with one or more sub-rectangles. The conditionals divide the sub-rectangle into three triangles: the condition sits in the triangle with the top border as its base, dividing the remaining space into two right triangles representing the two possible outcomes of the conditional. An example is shown in Figure 2.2.

The era of charts-like representation ended with the evolution of both hardware and software. To overcome the limitations of earlier approaches, new visualization techniques leveraged graphical user interfaces. In the 1980s, the main focus was on the visualization of the behavior of the program. An example is the visual representation of sorting algorithms [3]. Müller and Klashinsky presented Rigi [35], a tool to visualize system components and their dependencies in a graph model.

In the 1990s, more researchers gained interest in the field of software visualization. New approaches allowed for more fine-grained visualization. SeeSoft [13] represented up to 50,000 lines of code simultaneously. This tool used different colors to represent distinct statistics about the code.

Thanks to the use of versioning systems, the evolution of a system became a key factor of visualizations. The information can be collected and shown into new representations that allow the visualization of the lifetime

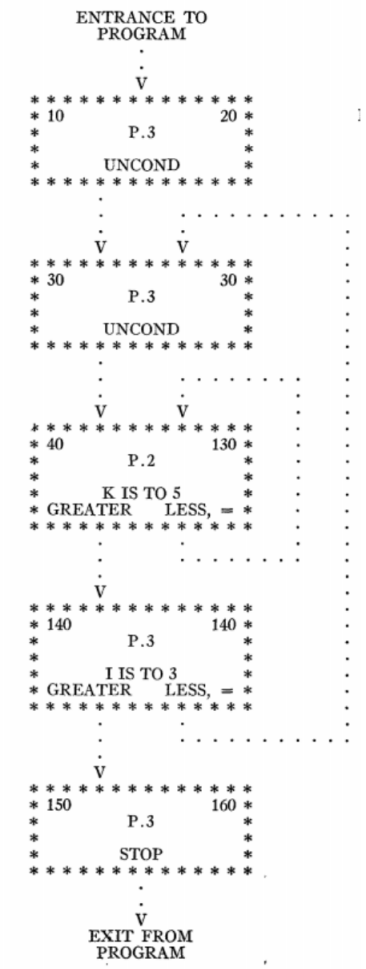


FIGURE 2.1: Flowchart [20]

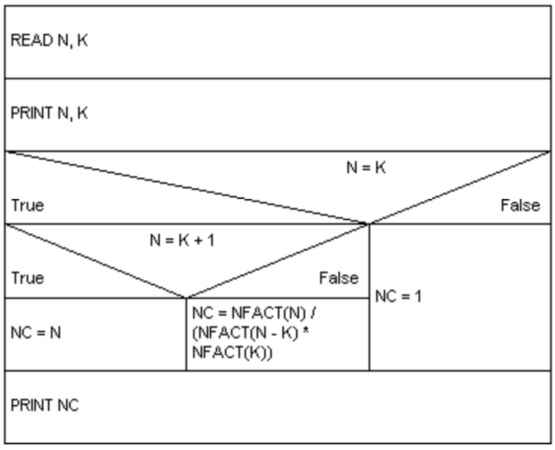


FIGURE 2.2: Diagram of Nassi and Schneiderman [37]

of a system. Holt and Pak created GASE: Graphical Analyzer for Software Evolution [21]. This tool visualizes the architecture of a system over time by mapping the age to colors.

In 1999, Lanza proposed a tool that analyzes the source code of objected-oriented systems, extracts several metrics from classes, methods, and attributes then visualizes them in different 2D forms such as graphs [28]. In 2001, he combined software evolution with software visualization presenting the Evolution Matrix [29]. As the previous one, this tool extracts metrics from the source code of each version of the program. The 2D visualizations shown in Figure 2.3 are composed of a matrix with the classes represented on the rows and versions on the columns. Each cell of a matrix contains a class of a specific version, represented as a rectangle with metrics mapped to its height and width. This visualization allows a complete understanding of the evolution of each class showing critical aspects as the change in size and their lifetime.

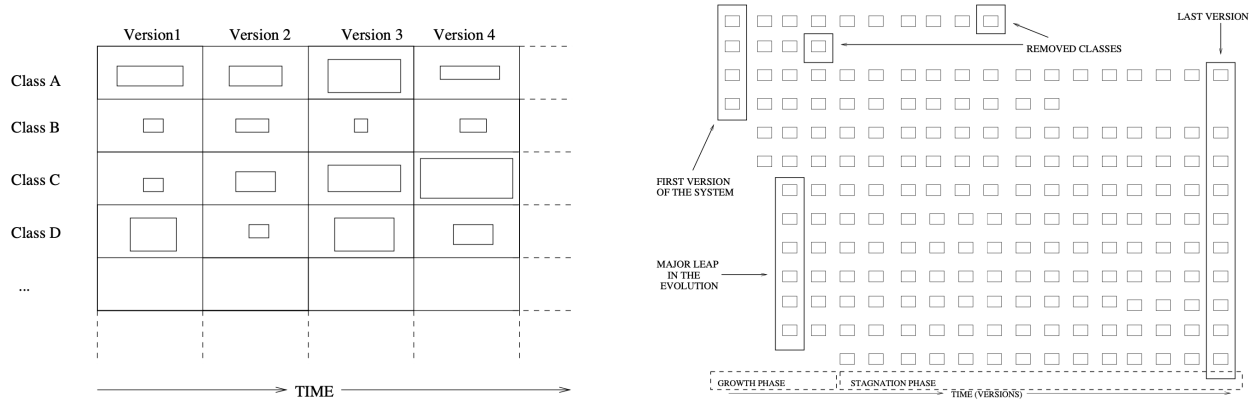
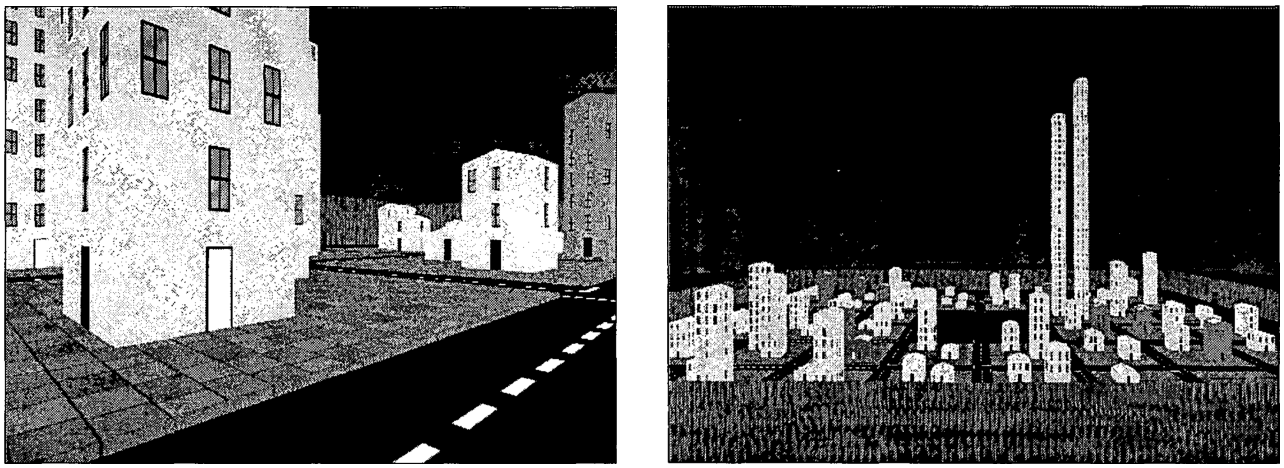


FIGURE 2.3: Evolution Matrix [29]

With the evolution of computers and their graphics, new approaches started to use the third dimension. It is the beginning of the 3D era. The addition of another dimension gives more space to the representations and adds new prospects. Gall *et al.* created a tool to visualize the structure of a system in both 2D and 3D graphs [17]. The third dimension is used to display the time. Colors are used to display the historical changes in the system. In 1995, Reiss created a 3D engine, called PLUM [43], to visualize the information of a system.

The field of software visualization became more interesting in the 2000s with the creation of dedicated venues. Greevy *et al.* presented TraceCrawler [19]: a tool for 3D visualization of the behavior of components, both statically and dynamically. Software analysis can extract a large amount of data that could not be displayed altogether due to the hardware limitations of those years. With the rising of the internet, web-based visualizations appeared. White Coat [32], presented by Mesnage and Lanza, is a web tool to visualize a Control Versioning System Repository in a 3D cube that contains the entities also represented as cubes.

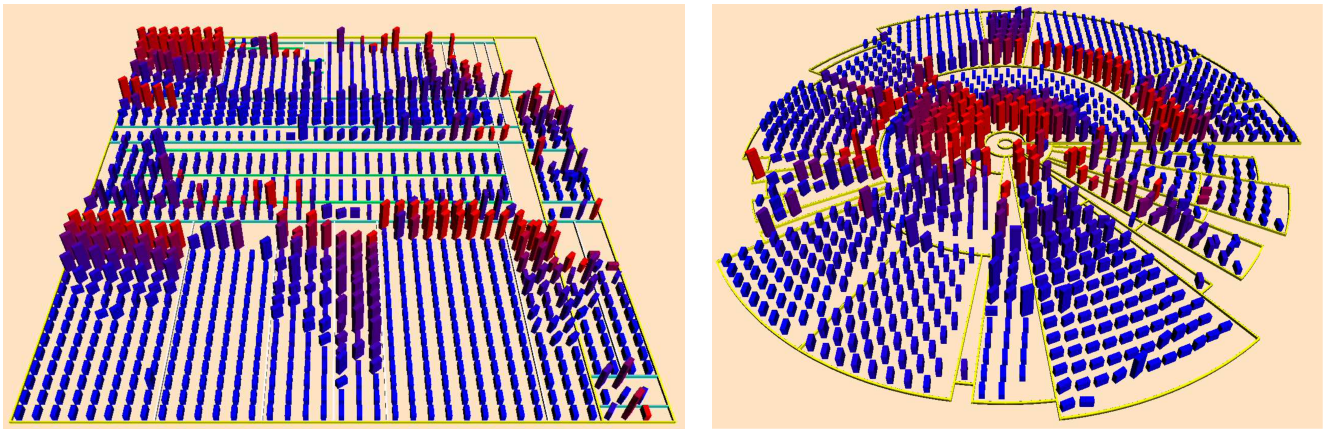
In 2000, Knight and Munro introduced a visualization based on the city metaphor [25]. This tool, called Software System, represents each class as a district with buildings made of methods painted differently depending on their visibility (public and private). Figure 2.4 shows the details of the buildings and an example of a city. Even though it displayed some irrelevant information, it shows the attempt to map source code to cities. This idea was used by Panas *et al.* who proposed a city visualization of retrieved



(A) Detail of buildings in a district (B) Overview of a district

FIGURE 2.4: Software System [25]

static and dynamic data [38–40]. In 2005, Langelier *et al.* presented different visualizations of 3D blocks that resemble a city [27]. Figure 2.5 shows two different visualizations of the same application composed of 1129 classes.



(A) Treemap technique (B) Sunburst technique

FIGURE 2.5: Visualization of the Quality of Large-scale Software Systems [27]

Wettel and Lanza, in 2007, revisited the city metaphor aiming to display software metrics meaningfully while trying to keep the layout of the city consistent with the information and giving viewers a sense of locality in the city [53]. The layout of the city can change in the different versions of the system. Hence districts and buildings can move around in the city. In 2010, Steinbrückner and Lewerentz used the city metaphor to visualize the evolution [47], shown in Figure 2.6. The city is visualized with time mapped onto the elevation of the hill on which buildings are placed, and the streets connect the different districts. The downside of this visualization is the way they handle space.

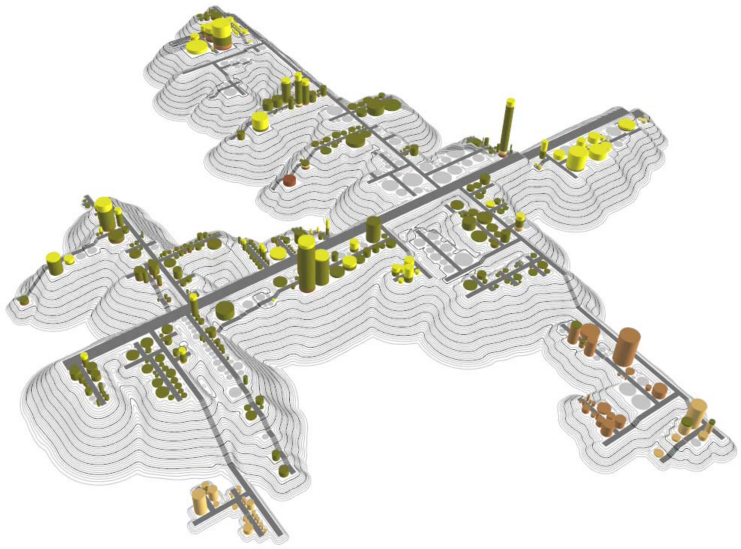
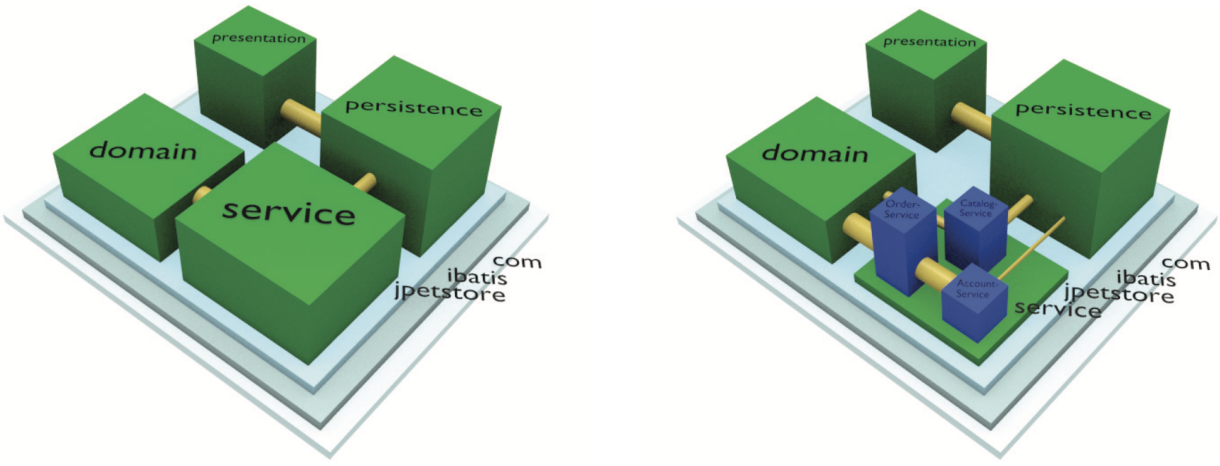


FIGURE 2.6: Representing Development History in Software Cities [47]

In 2013, ExplorViz [16] used a different approach to visualize live information by showing components in 3D blocks and their connections. Each package is represented as a district component, and each entity is represented as a building with height mapped to the number of instances. Buildings can be open to visualize the content, becoming districts. Communication among structures is visualized as pipes, figuratively streets, with thickness mapped to the number of requests. Figure 2.7 shows an example.



(A) Macro view visualization of four components

(B) Visualization of relationships with the service component opened

FIGURE 2.7: ExplorViz [16]

In 2012, Scanniello presented CodeTrees, where the software system is a forest made of classes shaped like trees with metrics mapped on the size, branches, leaves, and more [14]. This idea was used by Maruyama *et al.*, who presented CodeForest, where a system is also a forest with classes as trees, and there is a view to see the selected entity's source code at the bottom [31]. In 2015, Tymchuk *et al.* proposed Vidi [51] that analyzes the code quality in a 3D visualization of a city.

With the advent of Virtual Reality, researchers also started to leverage this technology for software visualization purposes. [15, 46].

2.1.1 M3TRICITY

M3TRICITY [41,42], which can be seen in Figure 2.8, is a language-independent 3D visualization tool for the evolution of large software systems presented by Pfahler *et al*. The authors use this metaphor to visualize the lifetime of a system. Their cities are complex, evolving systems made of blocks of buildings (source code) of many different sizes that are constructed, changed, maintained, and destroyed. The city comprises districts that represent packages. A district is composed of buildings representing classes, with metrics mapped onto the base and height. The overall score of the metrics extracted for each version of the system is shown in a chart at the bottom of the view. The layout of the city is consistent throughout the whole simulation. With rename operations, the buildings and districts are moved to a new location. The space dedicated to a class is reserved for the entire simulation.

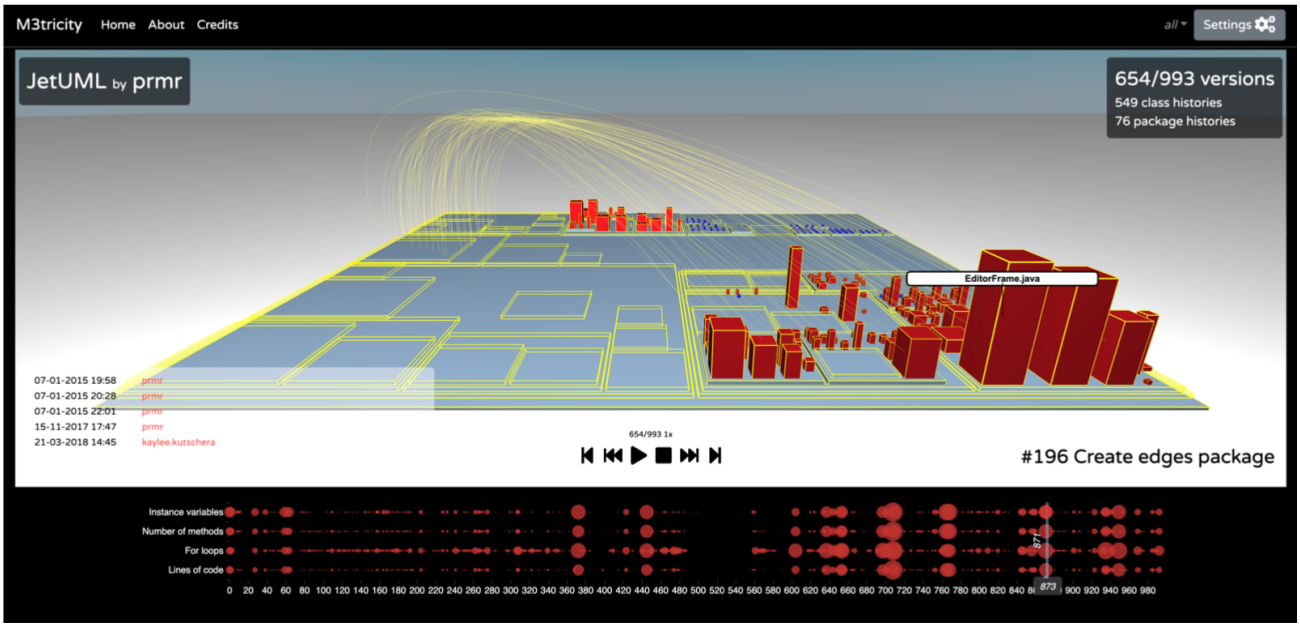
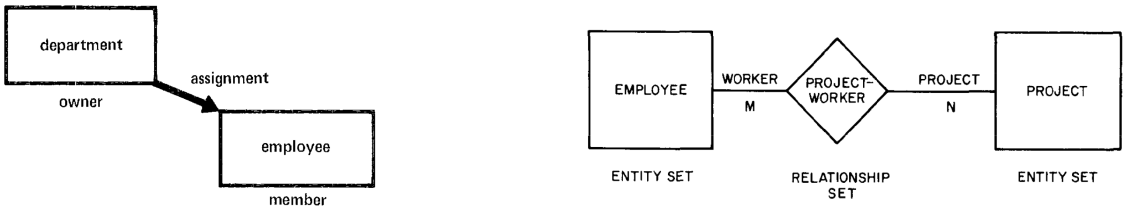


FIGURE 2.8: M3TRICITY [41]

2.2 Database Visualization

The first attempts to visualize database schemas date back to the 1960s, drawn by a human, shown in Figure 2.9. In 1969, Bachman presented a notation to visualize an "information model" defining both the terms and their representation [2]. His idea, shown in Figure 2.9a, represents each entity class in a rectangle box. The relation between two entities is shown with an arrow pointing to the entity that is a member of the other. The arrows are dashed if the relationship may not exist. In 1975, Chen presented a similar diagram with entity sets shaped like rectangle boxes and relationship sets like diamond-shaped boxes [8, 9], shown in Figure 2.9b. Entity-Relationship diagrams are still being used today.



(A) Bachman's diagram [2] (B) Chen's diagram [8,9]

FIGURE 2.9: Diagram representations

New visualization techniques proposed graphical user interfaces for databases. In 1980, Cattell presented an interface to a relational database to visualize in graphs the entities [5]. In 1982, Wong and Kuo presented a Graphical User Interface for Database Exploration, named GUIDE [54], which visualizes the schema of the database, or parts of it, as a network of entity and relationship types and aids the user in understanding it. In 1986, Davison and Zdonik presented a visual interface for a database with version management [12]. This system has an interface that allows the user to access the database graphically. The currently considered classes and entities are represented in a rectangle, and a hand points to the current database object. The same year, Bryce and Hull presented SNAP [4], shown in Figure 2.10. This interactive tool allows to design and visualize schemas for IFO models. The visualization, like most tools, is through an ER diagram.

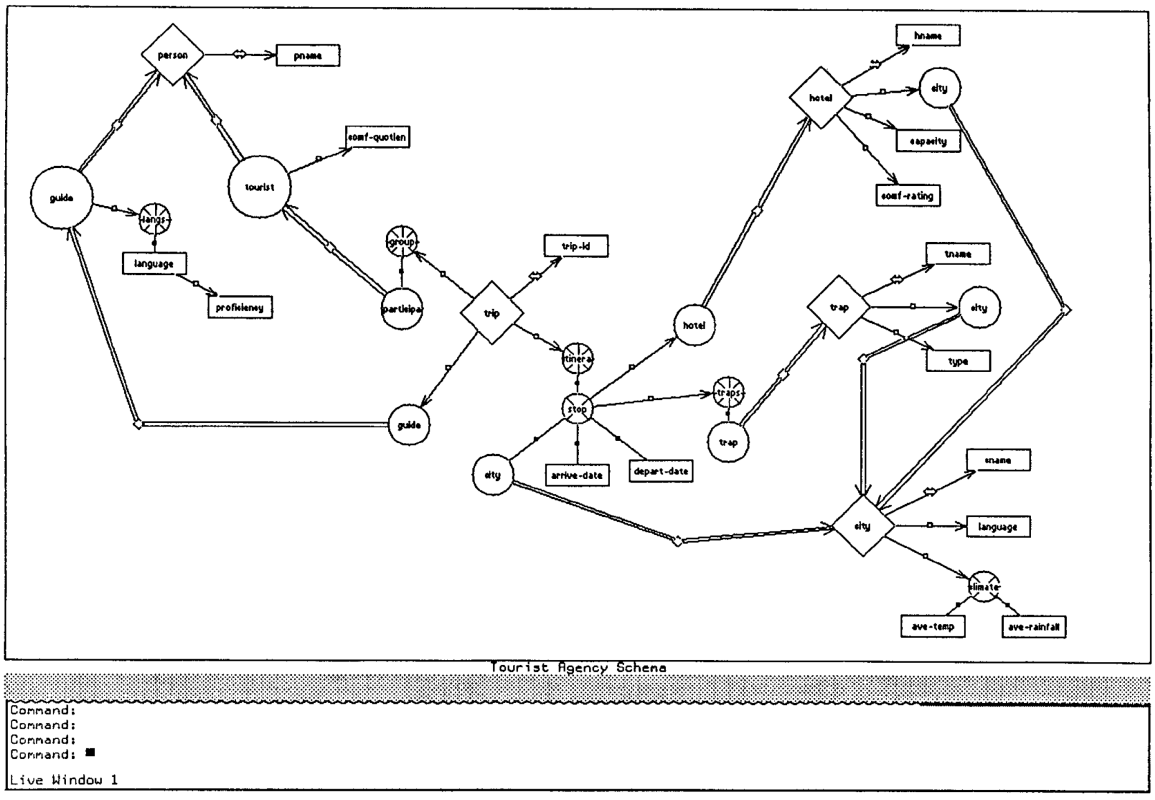


FIGURE 2.10: SNAP's Visualization of a database schema [4]

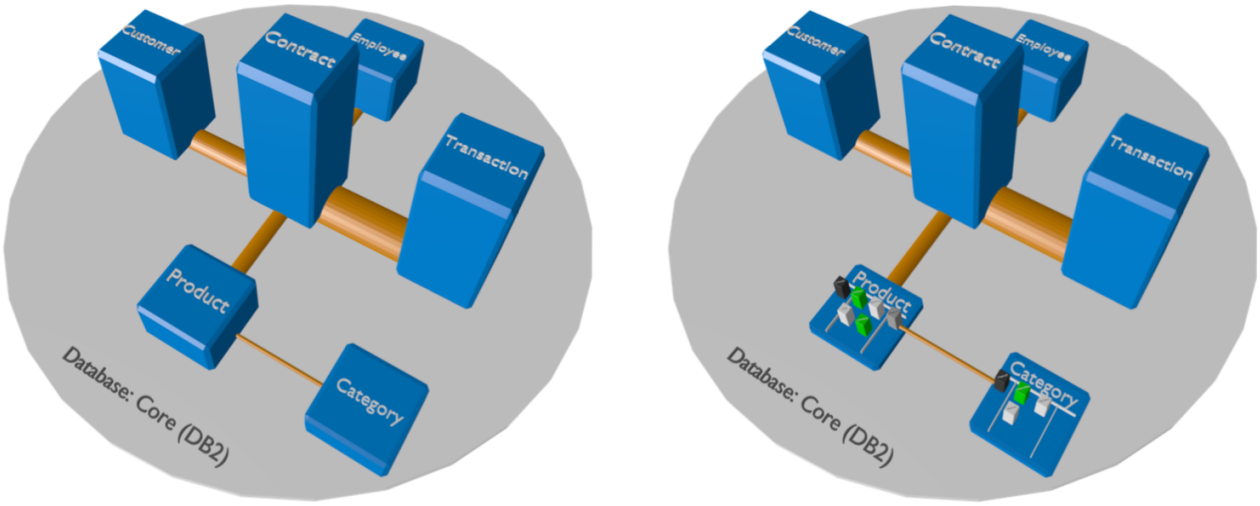
In 1987 Cattell and Rogers presented a tool to visualize databases as entity-relationship diagrams, called Schemadesign [44,45].

In the 1980s, researchers proposed ideas for possible graphical user interfaces for databases. Stonebraker and Rowe proposed an application program interface to a database management system [50].

In 1993, Agrawal *et al.* presented Tioga to visualize databases. This tool represents modules as boxes and connections as arrows which create a directed graph [48, 49]. In 1996, Aiken *et al.* presented an improved version called Tioga 2. The user interface was divided into data flow graph visualization, similar to the previous version, and a canvas for each program [1].

Database visualization has not been a popular field for years. Researchers have created tools to visualize queries, which allows for partial visualization of databases. In 1994, Keim and Kriegel presented VisDB to visualize datasets of multidimensional data by pixel in different representations [22–24]. The visualization colors the pixels depending on the relevance of the query given as input. In 2009, Chen *et al.* presented Schemr, a schema search engine that visualizes queries in graphs [6, 7].

In the 2000s, researchers have gained more interest in analyzing databases and their evolution. In 2008, Cortés-peña *et al.*, presented NakeDB [11]. This tool visualizes large datasets with four different representations: nodelink tree, circular, forced directed and radical tree. In 2019, Zirkelbach and Hasselbring presented RACCOON [55] which visualizes the internal structure of the database with a city structure. Each schema is represented as a circular district, and each table as a building with height mapped to the number of columns. A more detailed visualization shows columns as buildings on top of the table buildings, shown flat, with colored mapped to information such as primary key, foreign, and constraints. This city has streets that represent the communication between tables and, if opened, the actual columns. An example is shown in Figure 2.11.

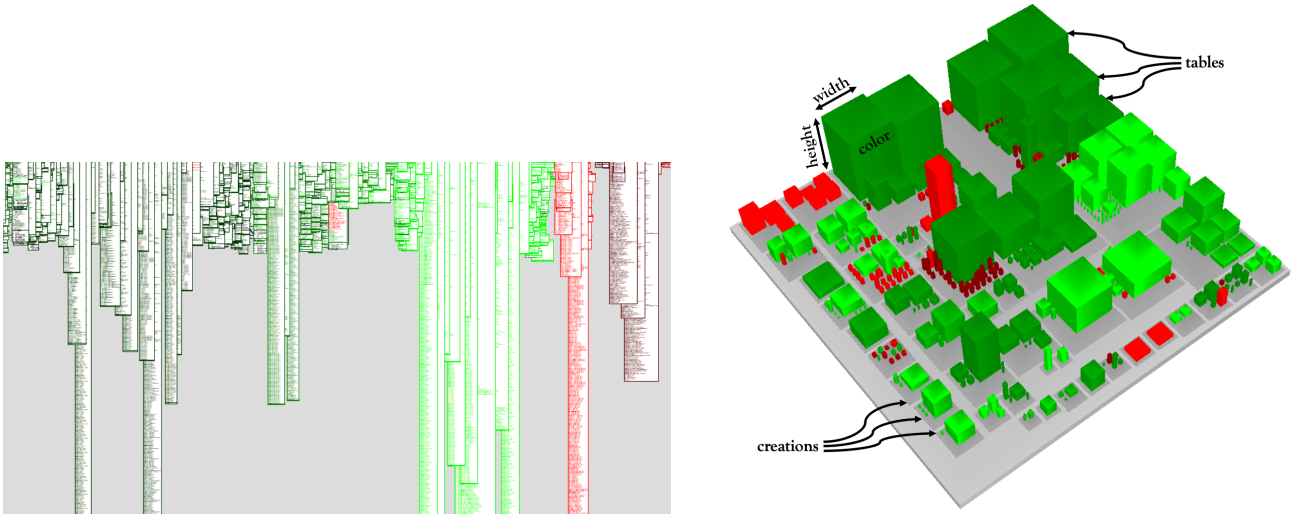


(A) Generic visualization of a database (B) Visualization with details on two tables

FIGURE 2.11: RACCOON visualization of a database [55]

2.2.1 DAHLIA

DAHLIA [33], shown in Figure 2.12, is a visual analyzer of database schema evolution, presented by Meurice and Cleve in 2014. This application visualizes the historical schema by extracting it from the SQL files of each version of the system [10]. The authors use different colors to show the age and liveness of each object. Green objects are still present in the last version, while red entities have been dropped. The 2D visualization, shown in Figure 2.12a, shows the schema as an ER diagram. This visualization is more suitable for small database schemas. The 3D visualization, shown in Figure 2.12b, is represented with the city metaphor, which is more suitable to visualize large database schemas. The districts of the city represented the time of creation. The buildings represented the tables with metrics mapped onto the base and height.



(A) 2D visualization of the historical schema

(B) 3D visualization of the historical schema

FIGURE 2.12: DAHLIA [33]

2.3 Software & Database Visualization

In 2016, Meurice and Cleve presented DAHLIA 2.0, a visual analyzer of database usage in dynamic and heterogeneous systems [34]. This tool goes beyond the visualization of schemas showing the database usage, extracting the accesses, and the location in the code. This tool uses the city metaphor to visualize two different cities (database and source code) side by side. When selecting a building of the source code city, the tool shows, in a side panel, the database access of the file. The tool highlights the selected building and the buildings of the database that the file access. An example is shown in Figure 2.13.

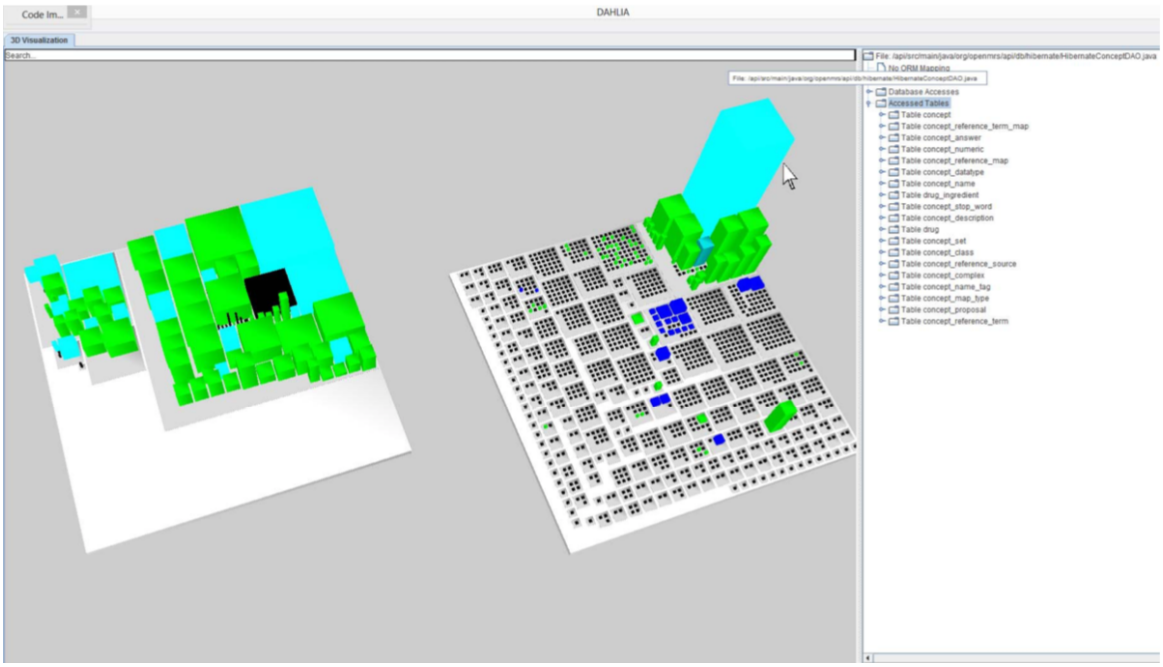
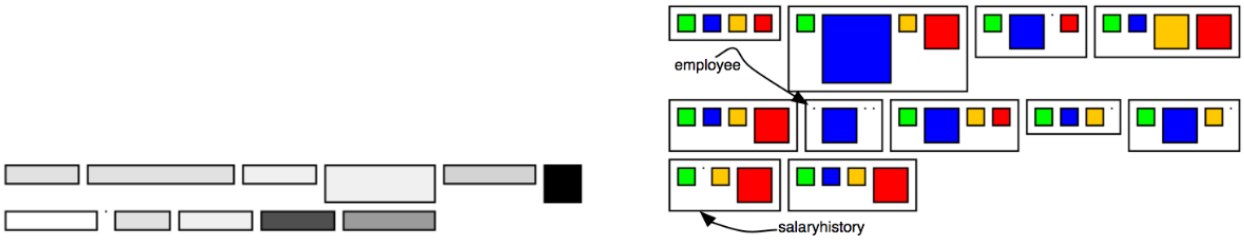


FIGURE 2.13: DAHLIA 2.0 [34]

In 2019, Marinescu presented a meta-model to visualize the relation of source code with the tables

of a database in enterprise systems [30]. The first representation showed each table as a rectangle. The width represented the number of statements in the source code. The height is the number of classes that perform operations on the table. The color is a gray gradient representing the number of columns in the table. An example is shown in Figure 2.14a. The second visualization represented the usages. Each table is rendered as a rectangle containing four squares that represent the type of operations. The side of the square represented the number of SQL statements. The color was mapped to the type of statement: `insert` is green, `select` is blue, `update` is yellow, and `delete` is red. Figure 2.14 shows the two representations of an industrial enterprise system.



(A) Table Accesses Visualization (B) Distribution of Operations Visualization

FIGURE 2.14: Visualization of Table Accesses in Enterprise Systems [30]

2.4 Conclusion

The techniques previously mentioned focus on visualizing either the source code of software systems or the schema of databases, but they do not consider both simultaneously. Their lives are strictly related to each other. A change in a database schema requires changes in the application code and vice-versa. Applications contain data within the systems, stored in files written in various formats. This type of information is versioned in the repository. Current approaches either treat these files as source code or filter them during the analysis.

As far as the city metaphor is concerned, M3TRICITY displays the evolution of a system with a resistant layout. DAHLIA 2.0 has a similar approach in the visualization of databases: it visualizes both components simultaneously. However, they are represented as two separate components in different cities. The interactions are visible only for one selected building at the time, which is not automatic as it requires human interaction.

We propose a novel approach to visualize a software system with its database and its interactions in the same city. The layout of the city is computed thinking of both source code and data. The visualization positions them efficiently to understand their nature and visualize their interactions.

Chapter 3

Software and Data Cities

In this chapter, we present our approach for visualizing software systems, databases, and their interactions using the city metaphor. Current approaches focus on visualizing either the code of an application or the data stored in a database. The software systems are composed of folders and files containing code. Additionally, they contain other types of files, which are usually not considered in the current visualization approaches that aim in visualizing only the software files contained in the systems.

However, database visualization approaches do not consider the whole software system. These approaches analyze the database itself to understand its structure and the data stored in it.

We present a novel approach to visualize software systems and their databases (i.e., source code and data) as evolving cities. The main goal of this approach is to represent the software system and differentiate the data sources (i.e., files and databases). We implemented our approach in a web-based tool named M3TRICITY 2.0, shown in Figure 3.1.

In Section 3.1, we describe our approach by explaining our model, the new metrics and the visualization. In Section 3.2, we describe the tool we implemented. We start by introducing the architecture of M3TRICITY. We continue explaining the web interface, the repository analysis, the database, our new visualization and, lastly, the improvement of the tool itself.

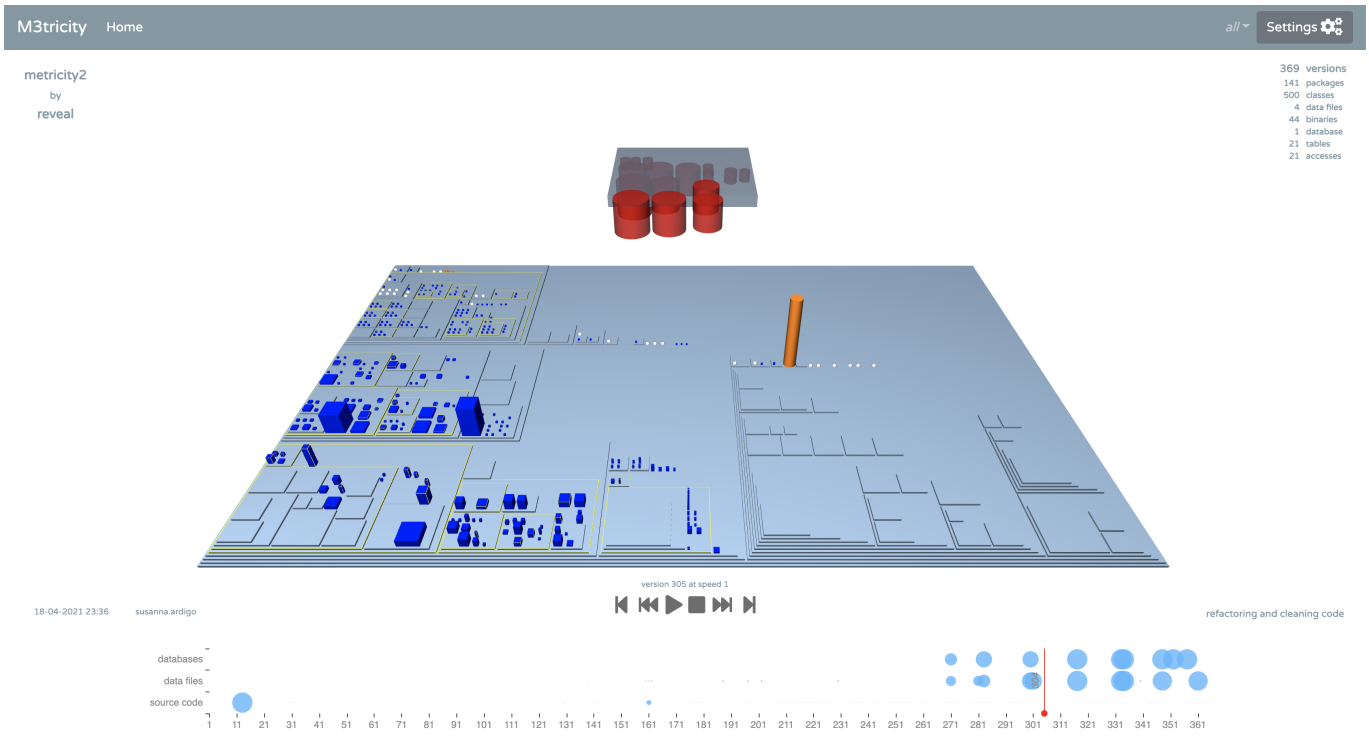


FIGURE 3.1: M3TRICITY 2.0

3.1 Approach

Software systems are composed of code in files that are contained in many nested folders. These systems can be very large and complex. The abstraction and intangibility of systems make them hard to understand. Moreover, modern systems are built around data, which fuels them and keeps them alive. This information is typically stored using different database technologies. The folder structure of applications can contain data in the form of different types of files. Our approach focuses on visualizing the software together with its data of different forms while trying to differentiate visually the data sources.

We use the city metaphor, which, in our opinion, is the most suitable type of visualization. Cities are enormous and complex systems created, changed, maintained, and, sometimes, sections can be destroyed, thus disappearing. Buildings are grouped into districts and sub-districts, dividing the city into neighborhoods. Our approach goes beyond the city metaphor, augmenting the city with new types of buildings. Each type of entity is mapped to a shape and a color. It confers the city with a completely new look. We decided to have a 3D visualization to place the user in a familiar environment that aids a fast comprehension of the system.

Our method aims to visualize the evolution of a city with a resistant layout, which keeps each entity at the same position throughout the whole simulation. We use M3TRICITY [41] as a starting point, a tool that visualizes the evolution of the software city. We make it extensible, and develop new features to visualize data.

3.1.1 Model

Our approach goes beyond the limitation of representing either software or data, having both parties in the same model. A new model is needed due to the new types of entities that the current models do not consider. We first need to make a distinction between the entity types. They can be classified based on their source (i.e., whether they are stored in a file or database), or based on what they represent (i.e., software or data).

- **Repository** is an entity representing a folder that contains subfolders and files that, all together, create an application or a project. It is stored on a server, managed by a versioning system and many copies reside on the computers of the developers.
- **Package** is an entity representing a folder contained inside a repository. Packages can contain other packages that, physically, are nested folders of the repository. The idea behind this entity is that it groups the content that can be identified by the package name. The main difference concerning the repository is that this entity contains a part of the application which cannot be considered a complete project.
- **Class** is a single file containing code written in a programming language. In object-oriented languages, such as Java, this entity represents a class as it is defined in the language. Depending on the language, a file might contain more than one class (e.g., Python), part of it (e.g., C++) or none. In languages, without the notion of classes (e.g., C), this represents a file containing code.
- **Data File** is an entity representing a file that contains data. Markup languages, such as XML, are used to define and store data in a semi-structured way. Some programming languages, such as JavaScript, provide a format to store objects, such as JSON.
- **Binary** is an entity representing a file that does not contain source code or data. Binary files can be of many different formats, they hold information, but it is not repository-related. An example can be an image: it can be used in the application, but it is considered a series of bytes rather than relevant data.

- **Database** is an entity representing an organized collection of data. Databases can be of different types depending on the technology used. The main difference between this entity and the data file is how the data is stored. Databases have a database management system that aids the programmer in creating, retrieving, updating, saving, and deleting the data.
- **Table** is an entity representing a collection of data stored in a database. It represents different collection types depending on the database type. In relational databases, tables represent structured data. Most NoSQL databases, such as document-store, represent data that is not necessarily structured.
- **Table Access** is an entity that describes when a class accesses a table. Formally it is an association class that links the two entities. It can represent different statements such as creation, retrieval, update, and deletion.

We developed a new model shown in Figure 3.2. We used as a starting point the model of M3TRICITY, shown in gray, based on the evolution model for software systems by Gîrba [18].

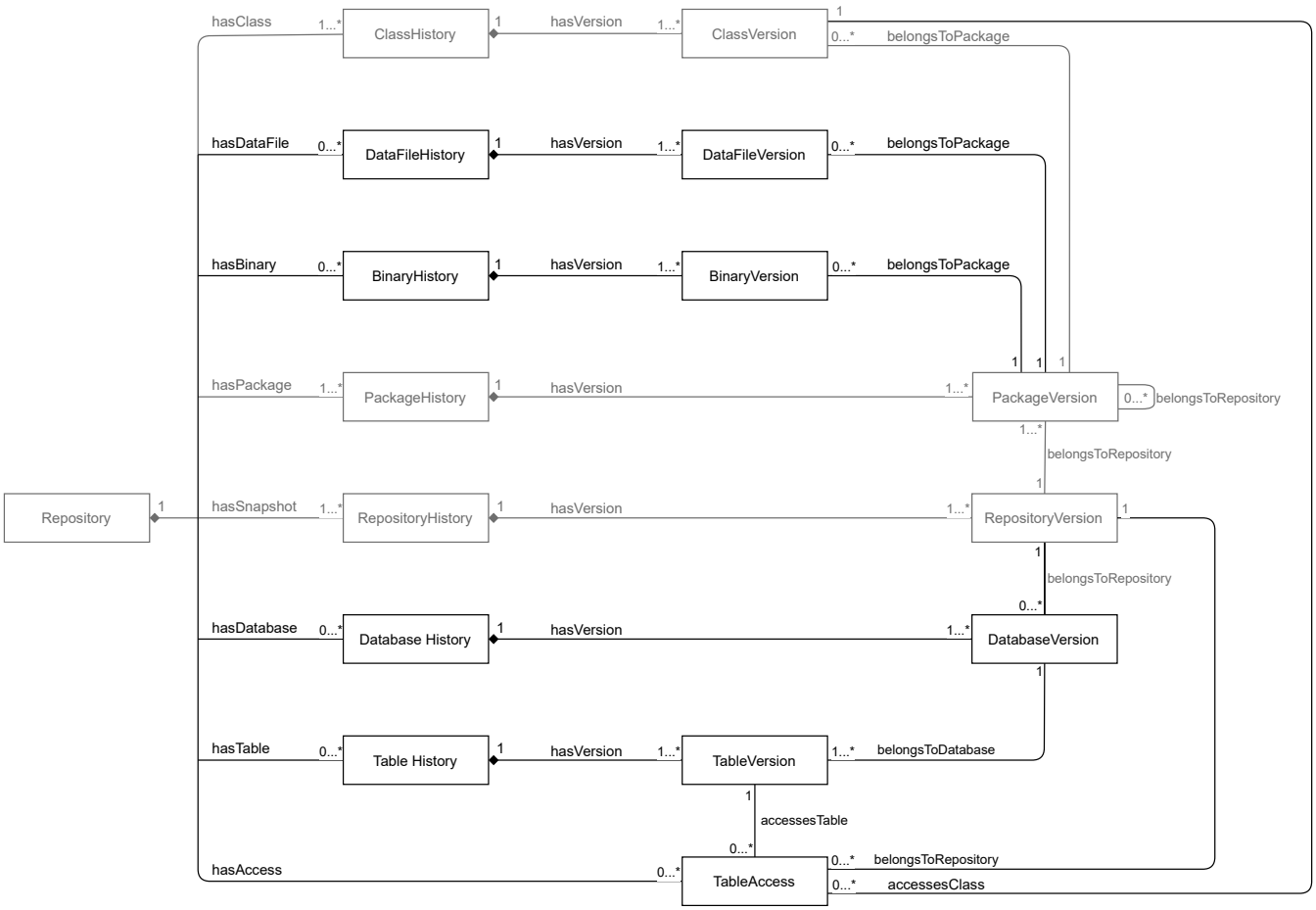


FIGURE 3.2: Model of M3TRICITY 2.0

The idea of the evolution's representation comes from the model of M3TRICITY and is then augmented with the new entity types. There are two important notions:

- **Version**: one entity of a snapshot.

- **History**: represents the history of an entity. It is a set of all the versions of an entity, starting from its creation, followed by its changes until its last change (or removal).

Each entity is modeled based on these two notions, with two exceptions. Repository has the concepts of version and history. As an addition, the generic `Repository` type represents the entire repository, containing all the histories of the entities, as well as the commits. `TableAccess` is a relation between a class and a table; thus, the only notion we have is its existence in a snapshot of the system.

Database

At the birth of a software system, there is no data, and this continues during most of the evolution of the system until it is deployed in production and tested with real users. At this point in the history of the system, the database can finally be filled with real-life data. The system needs to be tested before this stage can be reached. Developers can work around this problem with mock data, i.e., the creation and storing of fake information.

There are different possibilities about the timing of the introduction of the database. Some systems introduce it in the initial stages, while others tend to postpone it when the system has grown to the point when it makes sense to start using it. In the second case, the fake data can be stored in files before moving it into a real database. This introduces a different way of storing data: temporary data files. This moves information storage from the database to project files.

Many projects do not keep the database version controlled with the application code. The repositories do not contain information about it, with the only exception of its schema that, in some cases, is saved in a file. Databases can be, and normally are, versioned, but this happens on a separate private server. This is due to the restriction of the information. A live application with users may contain private and sensitive data.

The source code contains traces of the database accesses. Such code can be analyzed to find the location of queries, and then reverse-engineer the information needed. Our approach uses this technique to infer the database model. This process has to be done for each snapshot of the system to create the versions and histories of each entity. This information is not provided by a versioning system; thus, we need to infer the schema changes. Similarly to the changes in application code, we define the following change operations for databases:

- **ADD**: a new table or database is found for the first time, inferred from the source code.
- **MODIFY**: a table or database is altered. Regarding tables, the modification can be an addition, rename, or deletion of a column. Regarding databases, this operation concerns the tables.
- **RENAME**: an entity disappears in the same version and a new entity is found with the same information as the missing entity.
- **DELETE**: a table or database does not exist anymore. When we know that the entity is not alive anymore, we mark its deletion with the disappearance of its usage in the source code.

Git, in addition, has a **COPY** operation. Databases and tables do not have it, we consider copies as new entities. With this information added to each version, we can build the history of the tables and databases.

Data file

Systems have files that contain data, and we call them data files. While databases, in applications, are handled by the code, the data files are static files that the developer can change directly. This confers the developer the ability to directly create and edit the data stored in such files. These files are versioned with the source code.

Modern systems are built around data of many different types, which fits perfectly in a hybrid system with databases and data files.

Binary

Many binary files present in the repository were not meant to store data. The usage of these files can vary on the type and the goal of the application. Simple examples are the images. They can be used as input data for computer vision projects, and other uses. Our approach aims to display this type of file as a separate entity: it is part of the repository, but it is not considered source code or data.

3.1.2 Metrics

We define a set of metrics for each entity type in the model, we extract and store them with the model instance.

Table

Our approach considers relational and NoSQL databases. We decided to represent collections following the relational database model. A table is composed of columns and rows. The rows contain data entries, and the columns represent their attributes. We defined the following metrics for tables:

- **Number of Columns** (`COL`)
  This metric represents the number of columns of a table. We cannot know what information is being stored, but we can see the types and the structure. From this, we can find the names of the columns and count them. In practice, this metric shows the "extent" of the type of data.

- **Table Accesses** (`TA`)
  This metric represents the number of source code elements (i.e., classes) accessing the table. We do not have the certainty whether the piece of code that accesses the table is executed, we only count its existence. In practice, this metric shows the interaction of the application with the table.

Let us consider a generic database. Assume that there is a Java class that handles the saving of information in the tables and another class that retrieves the information. The two Java classes are accessing the database, which counts as table accesses. Suppose that both files access different tables, each access is counted individually.

Data file

Data files can store information in various formats. Our goal is to define metrics that can describe standard formats (e.g., XML and JSON) comprehensively. We examined different formats, extracted a common structure, and defined some notions, described below. The files tend to contain different types of information, stored in different forms depending on the usage. The content has repeating information patterns following the same structure. We call each type of structure an **Entity**. Individuals with similar structures are considered as different types. An entity is defined by its name and data structure, we called this notion **Property**. Entities can have sub-entities, creating nested structures. We considered the inner entity as a property of the outer entity. We defined the following metrics.

- **Number of Entity Types** (`ENT_TYPE`)
  This metric represents the number of the different types of entities that are contained in a single file. In practice, this metric shows the "breadth" of the stored information.

- **Number of Entities** (`ENT`)
  This metric represents the total number of entities that are contained in a single file. The number considers all the individuals of the different types of entities. In practice, this metric shows the "amount" of information stored.
- **Maximum Number of Properties per Entity** (`PPE`)
  This metric represents the number of properties of the biggest type of entity contained in a single file. In practice, this metric shows the maximum "extent" of the stored information.
- **Maximum Nesting Level** (`NESTED`)
  This metric represents the number of levels that the sub-entities have, being stored inside other entities. In practice, this metric shows the "nestedness" of the stored information.

Let us consider a JSON file containing JavaScript objects. Each JSON object is counted in the number of entities. Assume that the project that generated the objects has a main class and two subclasses. The instances of different subclasses are counted as different types. The properties, in this case, are the fields of the object, and the maximum number of properties per entity is the class with most of them. If one class has a field whose type is another object with a different structure, it is counted as another entity, as a different type of entity.

Binary

Binary files contain unstructured information. We define one metric to describe them: **size**. It can be displayed in different formats: byte, kilobyte, megabyte, and gigabyte.

Summary

Table 3.1 shows an overview of the metrics' names and identification codes.

Entity type	Name	Code
Class	Instance Variables	`DECL_STMT`
	For Loops	`FOR`
	Number of Methods	`NOM`
	Lines of Code	`LOC`
Data File	Number of Entities	`ENT`
	Number of Entity Types	`ENT_TYPE`
	Maximum Number of Properties per Entity	`PPE`
	Maximum Nesting Level	`NESTED`
Table	Number of Columns	`COL`
	Table Accesses	`TA`
Binary	Size	`BIN_SIZE`

TABLE 3.1: Supported Metrics

3.1.3 Visualization

M3TRICITY 2.0 is a web application for visualizing the evolution of software systems, data and their interactions. Our approach uses the city metaphor, which has been widely explored in previous years. We started from M3TRICITY, a tool that visualizes the evolution of a software system with a history-resistant

layout. This tool is based on CODECITY, a tool that visualizes software systems using the city metaphor. To understand our approach, we start by describing CODECITY, after that, we describe M3TRICITY. In the end, we discuss the core of the thesis: the novel visualization of software data, and their candidate interactions.

Software Systems as Cities

CODECITY is a tool to visualize software systems like cities. It is based on the idea of a direct mapping between software systems and cities.

The main elements of a metropolis are the buildings. They can be of different sizes, colors and created for various purposes. They create the setting for the existence of the city. These properties can be seen in a file that contains source code. Metrics allow the realization of **code-building** mappings.

Buildings are grouped into districts. A small town can be composed of a few districts, while a metropolis needs a more organized structure, grouping districts into bigger neighborhoods. This is the same notion of the folders of software systems. For this reason, we can have a **folder-district** mapping.

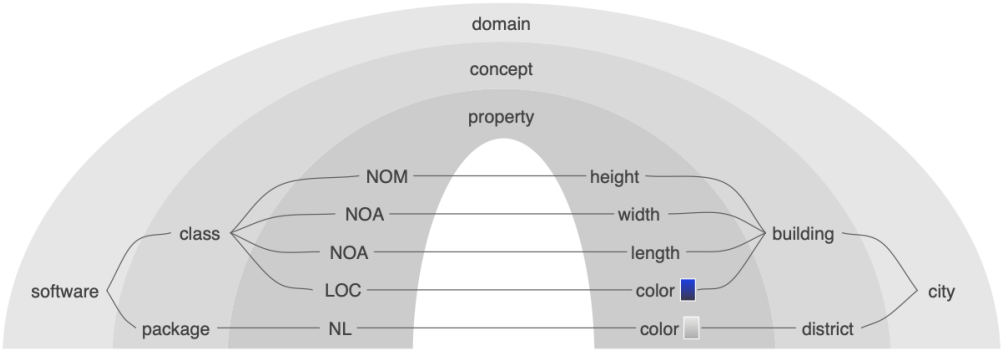


FIGURE 3.3: CODECITY mapping of a software system to a city [52]

CodeCity maps metrics to the size and color of each part of the city. The visual mapping reduces the overall complexity, aiding the user in understanding the structure and learn from the visualization. CODECITY uses the value of *Number of Attributes (NOA)* for the base of the building, *Number of Methods (NOM)* for the height, and *Lines of Code (LOC)* for the intensity of the color. The base of the district is rectangular, but its size depends on its content. The intensity of the color depends on the *Nesting Level (NL)* of the package. An overview of this mapping is shown in Figure 3.3.

The city uses a rectangle packing layout to achieve a scalable visualization of large-scale software systems. The algorithm optimizes the area of the city to be minimal. The core structure of the algorithm is a tree with the root being the main folder that contains the repository. Each node represents a district that can contain classes and other sub-districts. An example is shown in Figure 3.4. The layout is optimal, placing the biggest entities on the top left corner, adding progressively the smaller entities. It looks organized and confers harmony.

CODECITY is suitable for the analysis of one version of a software system, but not as efficient to visualize the evolution. The authors aimed at visualizing the best optimal city, analyzing one snapshot each time, without knowledge about other versions. Changes in the classes can result in different metric values and a different layout of the city. Buildings and districts can move around as the algorithm places them to minimize the city's area. In the end, it becomes very hard for the user to understand changes between versions.

M3TRICITY starts with the idea of CODECITY, addressing this limitation by adding the history-resistant layout. Its main idea is to compute the layout using the biggest size of each entity throughout its evolution. Each individual of the city has a reserved location throughout the whole simulation. The area of a building

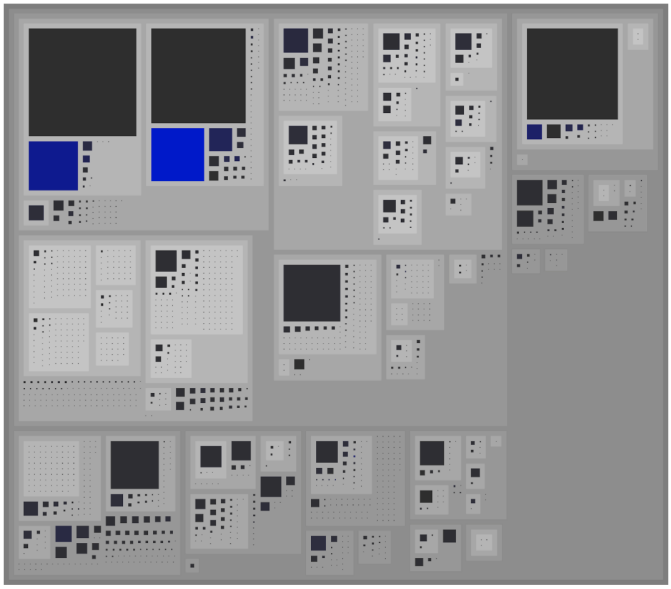


FIGURE 3.4: Layout of the CODECITY of ArgoUML seen from above [42]

or district is reserved for the entity. It is empty before the creation and after the deletion, and it is never reassigned to other entities. This approach creates a layout that in some versions can look sparse, but it facilitates the understanding of the evolution. Figure 3.5 shows an example of the comparison of the two different layouts of the cities. In the history-resistant layout, shown on the left, it can be seen that the packages have the same size in each version. The buildings have the same position, even with different sizes. The districts have empty space, which is used in other versions. In the rectangle bin-packing layout, shown on the right, the size of the package changes in different versions. The buildings are in different positions since the focus is on minimizing the wasted of space, which is little to not present. The history-resistant layout aids the understanding of the evolutionary process of the software system.

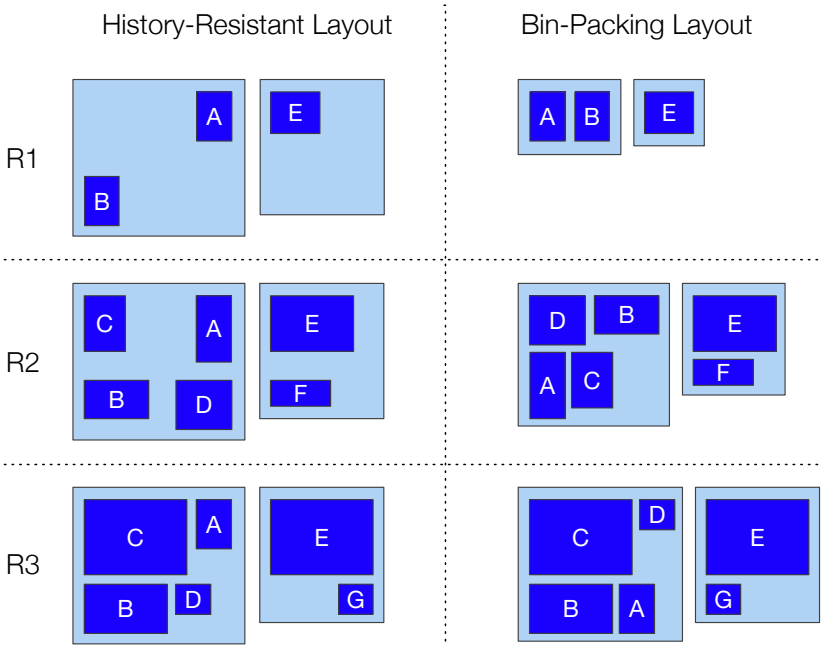


FIGURE 3.5: Layouts: M3TRICITY's History-Resistant vs. CODECITY's Bin-Packing [42]

Software and Data as Cities

We created M3TRICITY 2.0 augmenting M3TRICITY to visualize software data, in different forms, and their interactions. Before we introduce the new 3D representation of the entities of our model, we need to understand their source and role in both software systems and the city.

We have two types of entities that represent data: tables and data files. The first is contained in a database, and the second is part of the application files. We analyze a git repository that contains folders and files of code, data, and binary. These entities share a source, which is the git versioning system and should be visualized together. We do not have the real databases and tables, we create them by reverse-engineering the schema from the source code. For this reason, our city needs to be organized into two different parts based on the origin of the entities. Conceptually data files and tables have the same usage, which is the storing of information. For this reason, they need similar representations. The location in the correct part of the city will show the type of entity. With these distinctions, we can finally describe the new design with the augmented mapping.

A software repository has four types of entities: packages, classes, data files, and binary files. The two approaches introduced in the previous section visualize the first two types of entities with different roles and placement in the city. The **folder-districts** form the basement of the city and contain the **code-buildings**, which are the structures placed on top. These entities are represented with cuboids. The buildings have metrics mapped on them. The districts, however, have no metrics and are created based on the content, with a fixed height. Their colors are, respectively, blue and gray. An example of the 3D rendering of the mesh is shown in Figure 3.6.

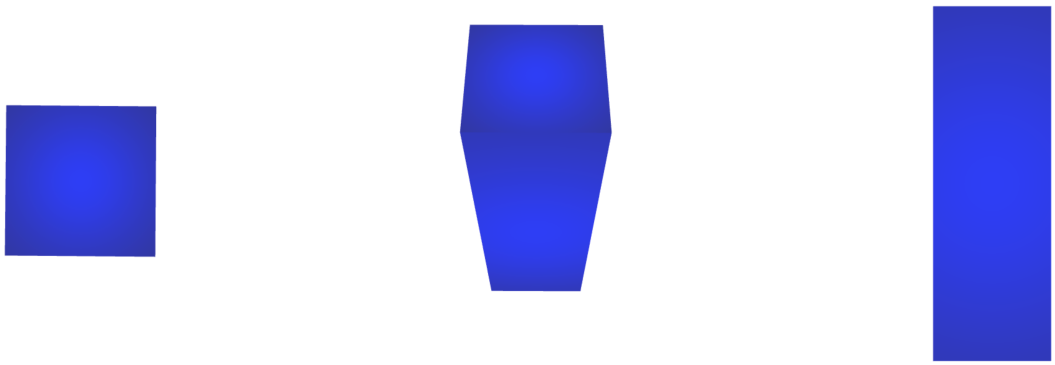


FIGURE 3.6: Code-building representation

The **data files** need a representation that expresses its difference while showing the values of the metrics. For this reason, we decided to use a shape with the same idea on the height but a different 2D shape for the base. We imagine data files as knowledge containers that flow and fuel the software. We created **datafile-cylinders**. The base is a circle, which allows the mapping of one metric on the diameter. The height, similar to the cuboid, is the second mapped metric. This new type of building is placed among the already existing code-buildings. The shape alone does not allow a fast understanding, new colors are needed. We color the data files in orange, so that they can be easily distinguished. An example of the 3D rendering of the mesh is shown in Figure 3.7.

Figure 3.8 shows a district composed of data files. The first two images show the view from the side, and the two following images show the view from the top. In M3TRICITY, the files did not have metrics,

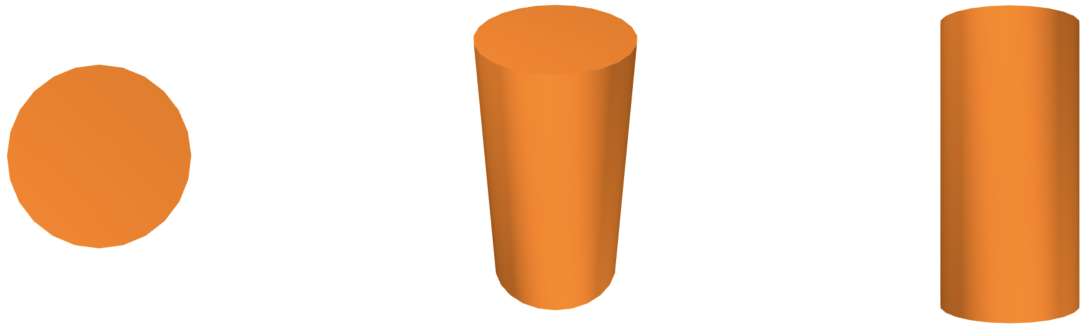


FIGURE 3.7: Datafile-cylinder representation

and were rendered with a minimum size. Our visualization has metrics, giving real information for their rendering. The new visualization aids a better understanding of the files.

The last type of entity that we can retrieve from the repository is the **binay**. To avoid the introduction

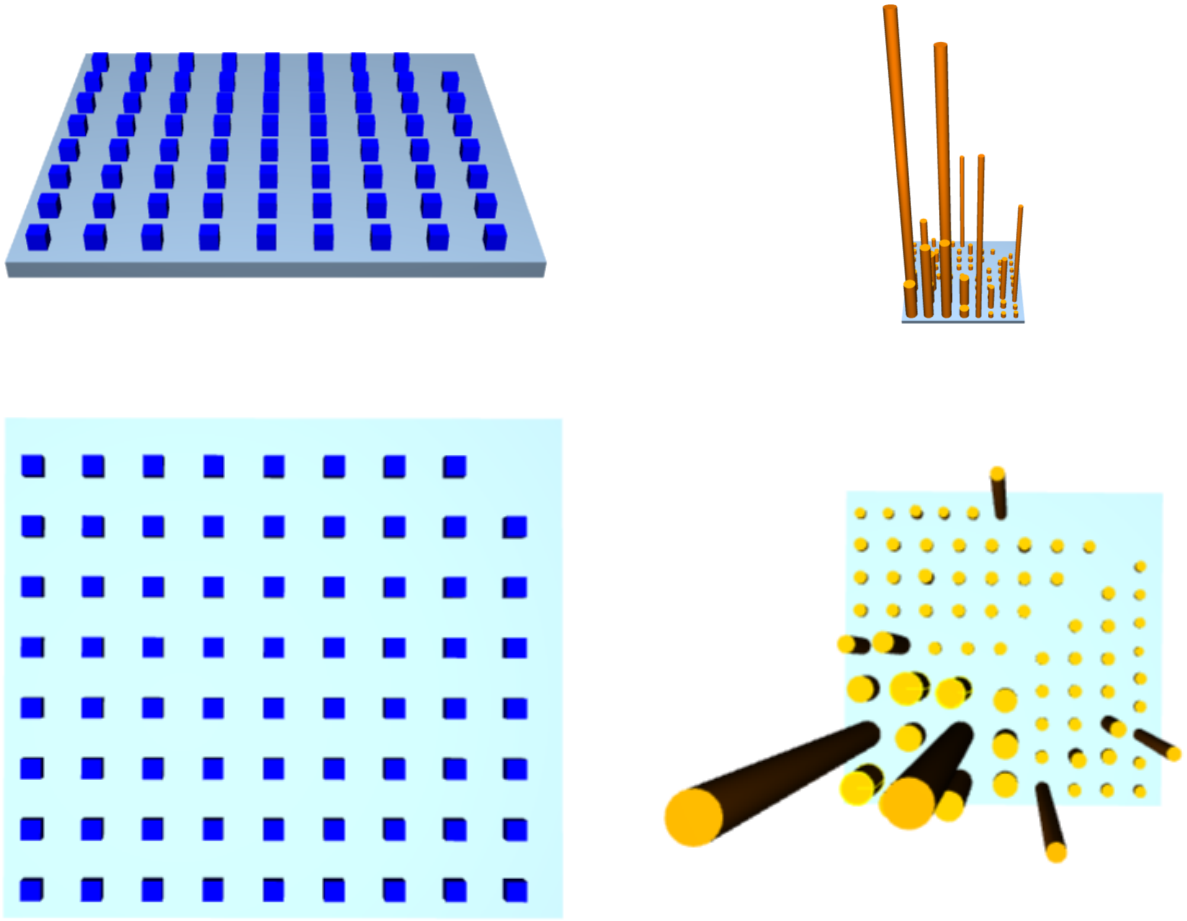


FIGURE 3.8: Comparison of data files visualizations

of new shapes, and create more noise in the visualization, we re-use one of the basics already present. We created **binary-hemispheres**. This shows that the file exists, and it is versioned in a repository but its use is less relevant for our purpose. We visualize each hemisphere without metric mappings. The color given to this entity is light gray. An example of the 3D rendering of the mesh is shown in Figure 3.9.

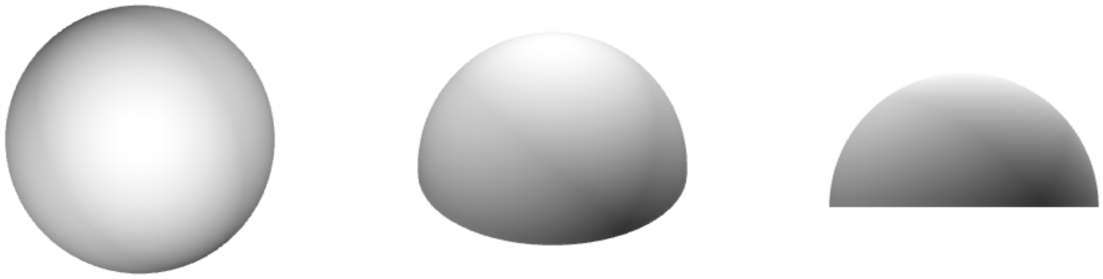


FIGURE 3.9: Binary-hemisphere representation

Figure 3.10 shows a district composed of binaries. The first two images show the view from the side, and the two following images show the view from the top. Our visualization renders all the binaries with the same size. The different shape gives a new look to the entity, aiding the user in a fast comprehension of the file type.

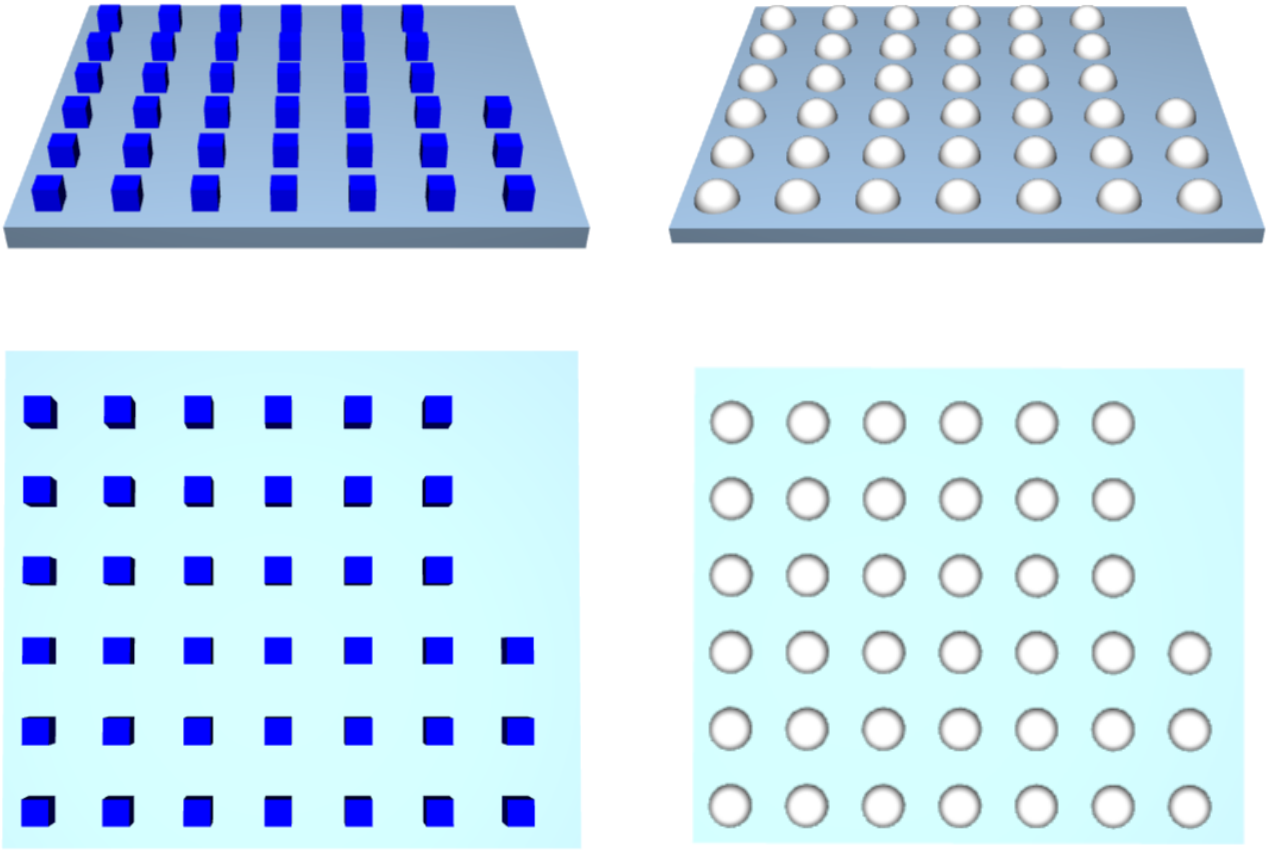


FIGURE 3.10: Comparison of binary visualizations

The second part of the city is pure data. The database, conceptually, is a container of tables that store structured data. For an easier understanding of the entities, we represent the tables with the same shape of the data files. We created **table-cylinders**. Its color is red, and the shape, as explained previously, allows the mapping of metrics on the diameter of the base circle, and on the height. The database is composed of different tables. It shares the same concept as the folders. For this reason, we shaped them as platforms, which is very similar to the idea of the districts. We call them **database-platforms**. An example of the 3D rendering of the table mesh is shown in Figure 3.11.

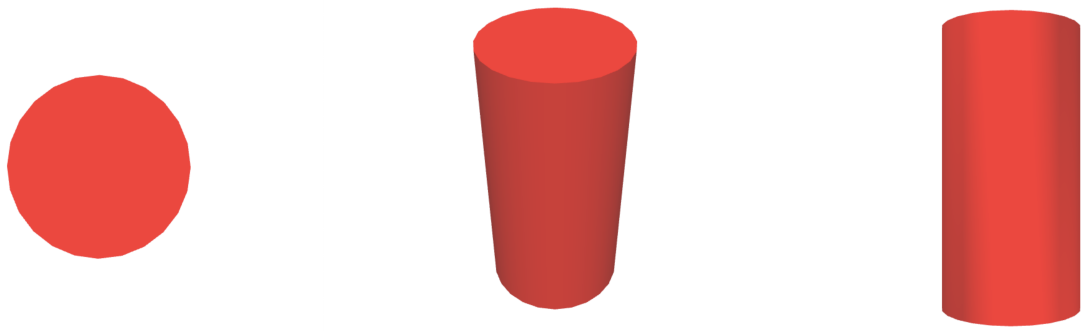


FIGURE 3.11: Table-cylinder representation

Databases are not physically present in the versioning system like the source files of Java classes. We create this part by reverse-engineering the schema from the source code. For this reason, we represent databases and tables with light transparency.

The last entity of our city represents the relation between source classes and database tables as **table accesses**. The relation links a class with the table that is accessed. We represent them as lines. To better understand their visualization, we first need to describe the representation of the city.

Each entity type has a set of metrics, which can be chosen as the value of the size of the base, height, or both. The conceptual difference of the entity types requires a distinction for the **mapping**. For example, *Number of Methods* of a source code file should not be equal to the data file *Number of Entities* and to the *Number of Columns* that a table has. They are conceptually different. The first represents the methods, which are collections of statements grouped to perform an operation. The second represents the amount of information stored in a file. The third represents the size of the data value set contained in a table.

A distinction is needed among the different types of data. The database and its tables are inferred from the statements found in the source code. We do not have knowledge about the amount of information stored, which could be zero. On the contrary, for data files, we have the content; thus, in this case, we have knowledge about the data, including its quantity. These two entity types are conceptually different and therefore need to be treated differently.

We introduced a multiplication factor for each side of each entity type. It customizes the visualization of the city by changing the mapping of the metrics. This allows the customization of the relevance of metrics within the same entity type. For example, a building can use the same metric for its base and height but with a different multiplication factor. A city with small classes and much information stored in data files can have a higher multiplication factor for source code, to increase their size, and a small factor for data files, to decrease their size. Vice versa, a city with big classes and little information stored in data files, can use the default value for the classes and increase the size of the data files with a higher multiplication factor. Systems with small databases can increase the size of the tables to visualize them bigger, with respect

to the software systems, and vice versa. Databases that use *Table Accesses* as a metric with high values can lower their multiplication factor to avoid having big sides of tables while using a small value for *Number of Columns*.

The city is composed of a basement, the districts, with buildings of different sizes, shapes, and colors. We separate the entities of the versioning system from the inferred entities: the elements physically present in the repository are represented by the city, and inferred elements of the database are vertically separated. To differentiate them visually, we decided that the new part is upside-down. The platforms are fixed at a specific location above/below the city and the cylinders grow vertically toward the city. We created two different types of cities with this concept.

The first type of visualization is the **City with Clouds**, shown in Figure 3.12. In this case, the city is below, with clouds above representing the database. The buildings grow to reach the sky, while the tables try to reach the basement of the city. In this case, the two components face each other. The space between them allows for a direct mapping of the interactions. The table access is shown with a line that starts from the bottom center of the table in the cloud and ends on the top center of the class that contains the query.

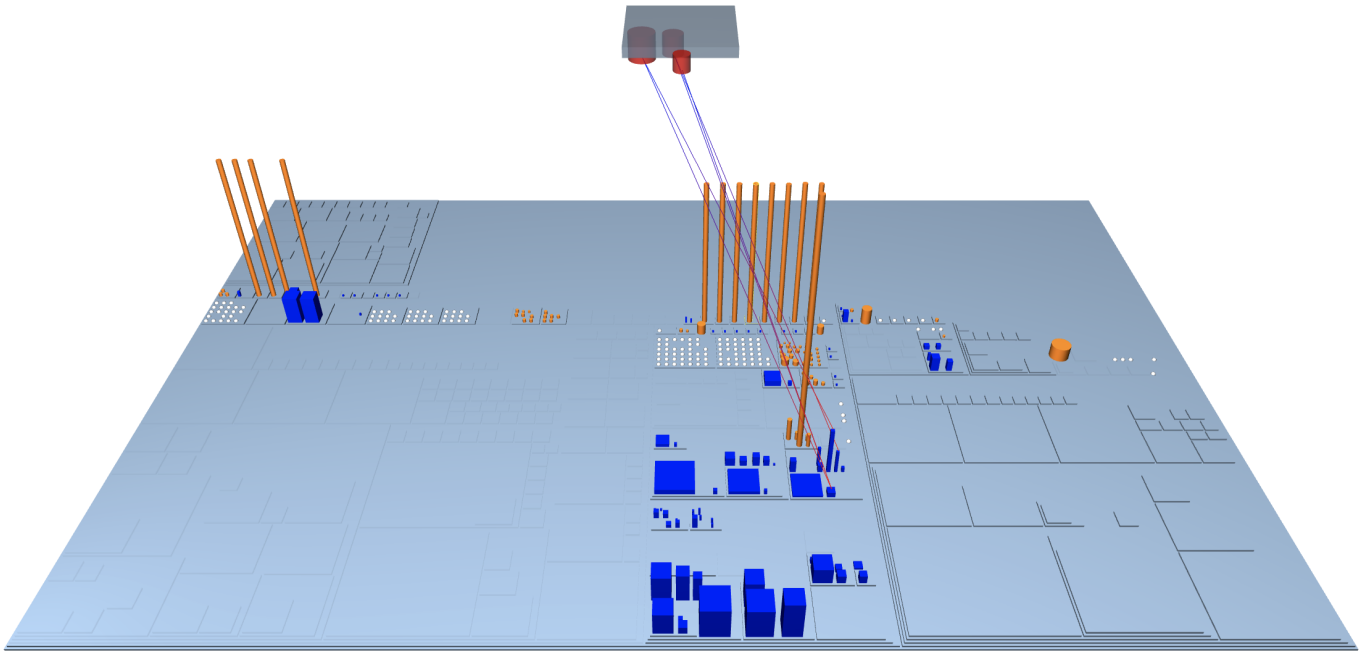
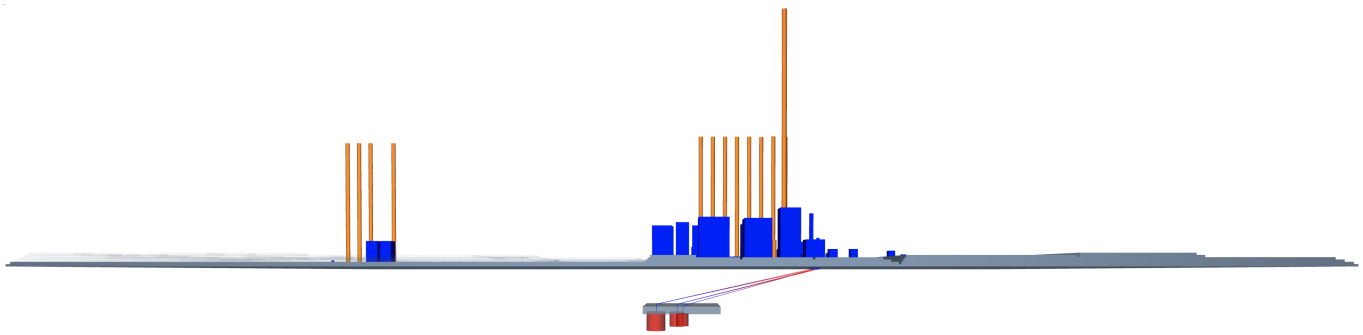


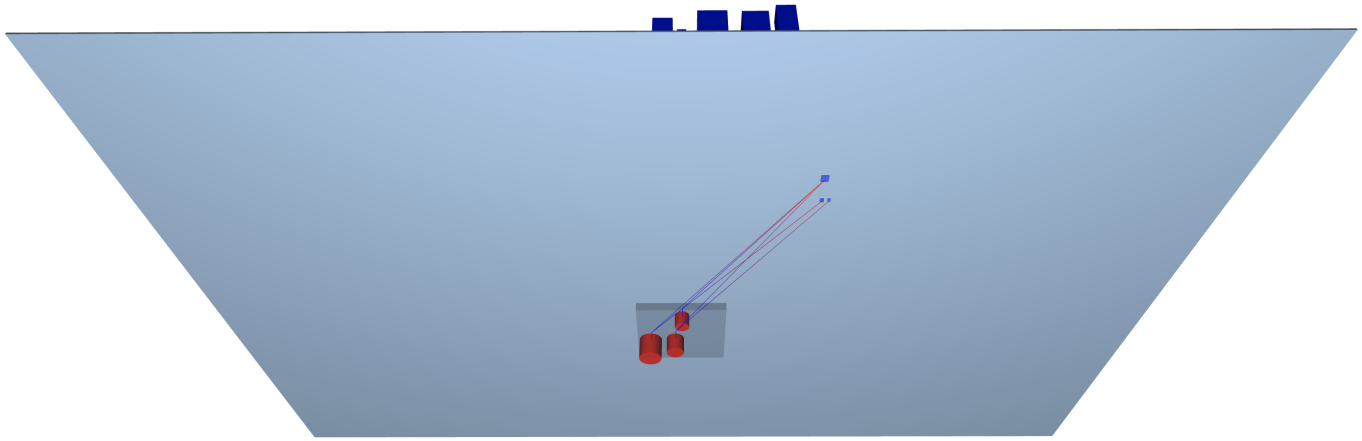
FIGURE 3.12: Cloudy City

The second visualization is the **City with Underground**, shown in Figure 3.13. In this case, the city is lifted to give space to the databases which are shown below. The city and the underground grow in opposite directions. They do not face each other. In this case, the visualization of the interactions is more complicated than in the previous city. To connect the buildings of the classes and the tables they are using, we create a basement of the classes below the city. It is see-through as the underground. With this added block, we can link the basement class with the top of the table. Figure 3.13a shows the side view of the city. Figure 3.13b shows the underground, composed of the database platforms and the basement of the classes that access tables.

Our approach features also a visualization that shows only the city constructed from an application, without databases, tables and accesses.



(A) Side of the City



(B) Underground view

FIGURE 3.13: City with Underground

3.2 Implementation

In this section, we discuss design and implementation details of M3TRICITY 2.0. We start by describing the architecture of M3TRICITY and how we made it extensible. Then we describe the implementation of our analysis, metrics and visualization. We also compare the two tools, showing the improvements.

3.2.1 M3TRICITY

We decided to use M3TRICITY as a starting point, made it extensible, and augment it to extract, analyze and visualize data. The architecture of M3TRICITY consists of a backend and a frontend.

Backend

The backend is implemented using Java 11[1] and Gradle 6.[2] It is built using Spring Boot,[3] a framework for developing web applications. It is composed of three main modules.

The first module is **History-Builder**, it analyses the project and creates a Java object representing it. It consists of the following three sub-modules:

- **Repository Downloader**: clones locally in the `resource` folder a publicly accessible GitHub project.

[1] https://docs.oracle.com/en/java/javase/11/docs/api/
[2] https://docs.gradle.org/6.3/release-notes.html
[3] https://spring.io/projects/spring-boot

- **History Model Builder**: analyses each snapshot of the system in chronological order, and creates the history of packages and classes, which consist of the different versions.
- **Metric Extractor**: runs srcML[4] to creates an XML representation of each class and then processes it to extract the metrics.

The second module is **View**, which prepares the data for the visualization of the software system requested by the user. It consists of the following two sub-modules:

- **Layout Computing Engine**: computes the overall history-resistant layout of the city and updates the view to other versions.
- **Visual Models**: contains the models of the meshes in the city.

The third module is called **Endpoints**. M3TRICITY uses two technologies to communicate with the frontend: REST API and WebSockets. The first uses the GET and POST requests of HTTP methods to access and insert data. For example, the analysis of a new project is a POST request with the URL. The list of available repositories is a GET request of projects both analyzed and being analyzed. In general, each entity of the city has endpoints to retrieve the information stored in the backend. WebSockets are used for the layout of the city. There are two PUBLISH endpoints: start and update. The first starts a visualization asking the backend to compute the layout of the city and retrieve the basic information of the project. The second updates the view of the city, it is used to progress further in the history, go back and change the settings of the visualization.

Frontend

The frontend is implemented in TypeScript[5]. It uses the open-source framework Vue.js[6] for the user interface. The skeleton is created with Bootstrap[7], which is an open-source CSS framework to create responsive blocks. The 3D visualization is created using Babylon.js[8], a real-time 3D engine using a JavaScript library for displaying 3D graphics in a web browser. It is composed of three modules.

The first is **View Manager** which manages the view of the city. It is composed of the following two sub-modules:

- **Canvas Builder**: creates and updates the Babylon.js canvas. It handles the creation, modification, and deletion of the meshes, which represent the entities of the software system. They are identified with the id assigned during the analysis in the backend. This sets all the lights, the camera, and makes the meshes clickable.
- **Time Scheduler**: schedules the simulation of the evolution. When a new version of the city is requested, the fronted waits for the response of the backend and applies the changes by animating the meshes. This requires time. This module prevents overlapping animations by waiting until the completion to proceed with the new version if the evolution is playing.

The second module is the **Graphical User Interface**, also called **GUI**. At first, the user is presented with the **Home Page**. It is composed of an input field to start the analysis of new repositories, and a table with the list of repositories that have already been analyzed or are currently being analyzed. By clicking on the repository name, the user is brought to the **City View**, the second and most important page. At the center of the screen, the user is presented with the 3D visualization of the software system through the city

[4] https://www.srcml.org
[5] https://www.typescriptlang.org
[6] https://vuejs.org
[7] https://getbootstrap.com
[8] https://www.babylonjs.com

metaphor. At the top of the center, on the left, the owner and the name of the project are shown. On the right, there is the number of the current version, the total number of versions, the number of how many histories of classes and packages. In the bottom center of the screen, the buttons control the evolution: start, play, previous and next version, slower and faster multiplier. On the right, there is the message of the commit. On the left, there are dates, times, and authors of five commits. The information representing the current snapshot is located at the center and highlighted with bigger font size. At the bottom of the screen, a timeline shows the evolution, with the snapshots on the x-axis and the metrics on the y-axis. On the top, there is the menu bar with the home button to go back to the list of repositories. On the top right, there is the settings button, which opens a new page on top of the visualization. This allows the user to customize the visualization.

3.2.2 Model

Entity classes of M3TRICITY 2.0 extend the `MetricityBaseEntity` class. It is composed of the field `id`, which is created randomly on the initialization of a new object, with prefix `M3-`.

Versions

The first part of the model consists of the **Version** entities. The diagram is shown in image 3.14. The

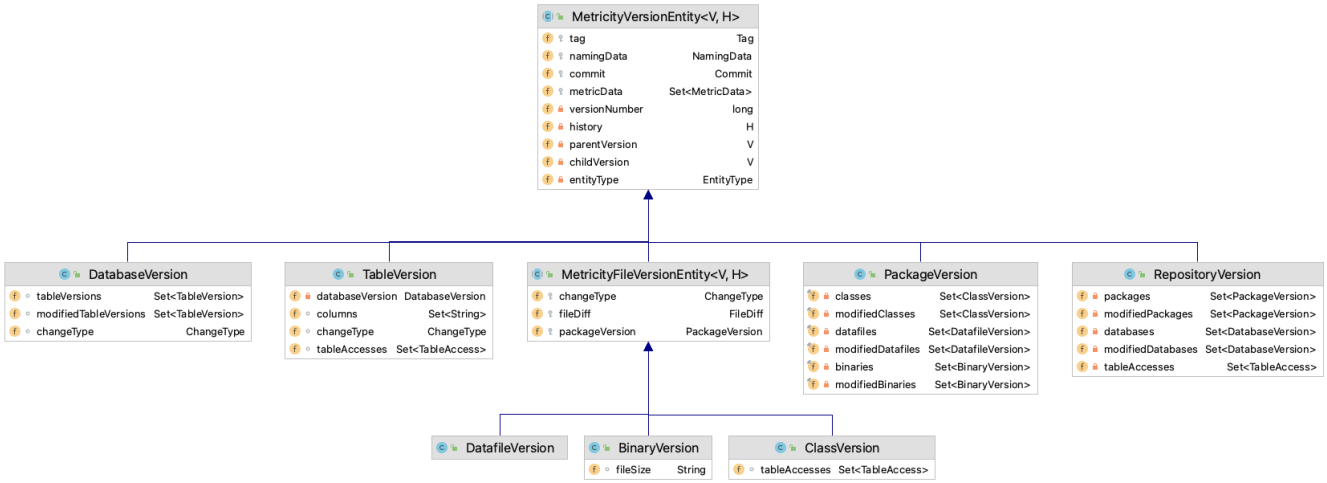


FIGURE 3.14: Model: Version Entities

superclass `MetricityVersionEntity<V, H>` defines the fields of the version entities. The generic types define the entity type, with `V` being the version and `H` being the history class.

- `tag`: is an object representing a set of strings that describe the object. In file entities the tags are created from the path, tables are created with the names of the columns, the container entities have the sets of the objects that they contain.
- `namingData`: represents the current and original name of the entity, and its type.
- `commit`: is the commit object that introduced the version of the entity.
- `versionNumber`: is the number of the version, starting from 1.
- `history`: is the object representing the history of the entity.
- `metricData`: is a set of MetricsData objects that contain the type of the entity, the metric, and its value.

- `parentVersion`: is the previous version of the entity, if it exists.
- `childVersion`: is the next version of the entity, if it exists.

The **database** is implemented in the `DatabaseVersion` class. It extends the superclass with the addition of three fields. The entity contains tables represented as a set of TableVersions named **tableVersions**. The field `modifiedTableVersions` is the subset that contains only the tables that have changed from the previous version. We introduced the change type to versions of the database, stored in the object `changeType`.

The **table** is implemented by the class `TableVersion` with the addition of four extra fields. The field `databaseVersion` is the object that the table belongs to. A table is composed of a set of columns, whose names are saved in a set named `columns`. We introduced the change type to versions of tables, stored in the object `changeType`. The field `tableAccesses` is a set of objects representing the relations of classes accessing that table.

Classes, data files, and binaries are all entities that, physically, are files of a repository. For this reason, we created a common superclass to describe the common structure. `MetricityFileVersionEntity` has the three following fields.

- `changeType`: describes the type of change that the entity has in the snapshot.
  It can be ADD, MODIFY, COPY, DELETE.
- `fileDiff`: represents the difference of the file compared to the previous version.
- `packageVersion`: is the object representing the parent package of an entity.

The **data file**, implemented in `DatafileVersion`, does not have any extra field compared to its superclass. The **binary**, implemented as `BinaryVersion`, has an additional `size` field, a string representing its size rounded to the closest unit of measure. `ClassVersion` represents a **class** with the addition of a set, named `tableAccesses`, that represents the relation of that class accessing tables.

A **package** is implemented in `PackageVersion`. This type of entity can contain classes, data files, and binaries. For each category, we have two fields. The first is the set of the objects that belong to the package, the second is a set of the objects modified in the current snapshot.

The **repository** is implemented in `RepositoryVersion`. It contains the set of packages and databases, as well as their subsets containing only the individual modified in the current version of the snapshot. In addition, it has the set of the table accesses relations.

Histories

The second part of the model is the **History** of each entity. The diagram is shown in image 3.15. The

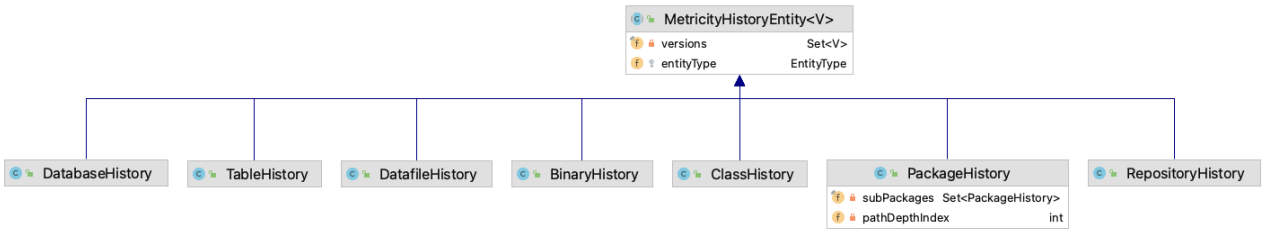


FIGURE 3.15: Model: History Entities

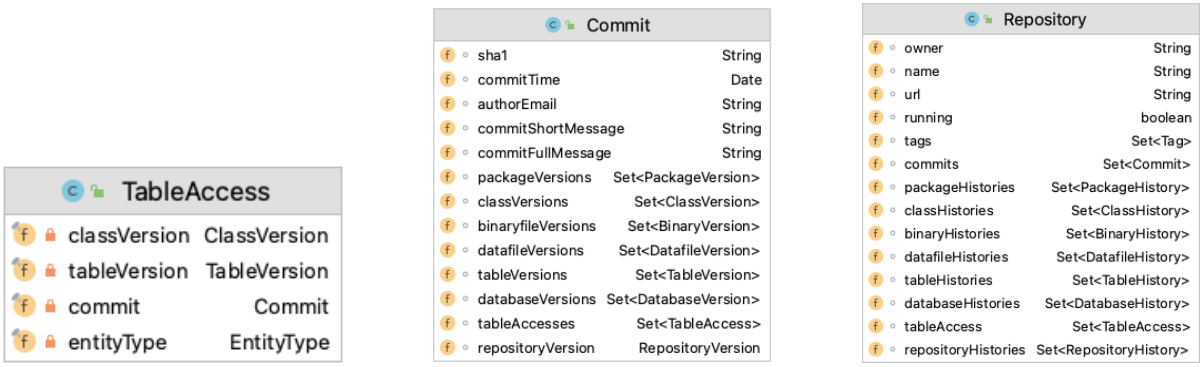
superclass **MetricityHistoryEntity<V>** defines the structure. It uses the generic type of the version. It has two fields:

- `versions`: is the set of version objects that represent the life of the entity
- `entityType`: represents the type of the entity.

Each of the seven entities is implemented in a subclass. `PackageHistory` is the only entity that has extra fields. It has a set, named `subPackages` that represent the packages under the package itself. The second field represents the level of nestedness of the package and is called `pathDepthIndex`.

Other classes

The third part of the model is composed of single classes that do not represent the version or history of an entity(see Figure 3.16).



(A) `TableAccess` (B) `Commit` (C) `Repository`

FIGURE 3.16: Class diagram of `TableAccess`, `Commit` and `Repository`

The relation of a **class accessing a table** is implemented in `TableAccess`, shown in Figure 3.16a. The class has two fields representing the two objects that form the relation, the object representing the commit that created the snapshot that the relation was found in, and the entity type.

The **commit** is implemented by the class `Commit` and has thirteen fields. The class diagram is shown in Figure 3.16b. The field sha1 represents the string that identifies the commit. The date and time of the creation are saved in the object `commitTime`. The fields `commitShortMessage` and `commitFullMessage` represent the message that describes the commit. The first version is truncated to be shorter. Each of the seven entities has its own set that represents the version object that the commit introduced. The relations of the table accesses found in the snapshot of the commit are saved in a set.

The **repository** is implemented by the class `Repository`. The class diagram is shown in Figure 3.16c. The owner, name, and URL are saved as strings. The database type (with the dialect) is saved in the field `databaseType`. The boolean `running` is true if the analysis is currently taking place, in the end, it becomes permanently false. The repository has a set of tags, which represent all its entities. There is a set of commit objects. Each of the seven entities of our model is saved in its own sets of histories. The relations of the table accesses found in the snapshot of the commit are saved in their set.

3.2.3 Home Page

A typical use case of M3TRICITY 2.0 starts when a user opens the webpage. The home page, shown in Figure 3.17, has the logo in the center of the page followed by two parts. The form is composed of a search bar to insert the URL of an existing public GitHub repository. The user can choose as advanced setting the database type and dialect for the analysis. Our platform has the following types: Apache Impala, MongoDB, MySQL, SQLite and no database. With the inputs filled with valid information the request to start the analysis can be confirmed and sent to the backend. The second part is the table of repositories.

The columns show the name of the project, the owner and database type. The top rows show the projects ready to be visualized, while the bottom rows are currently being analyzed and cannot be visualized yet.

FIGURE 3.17: Home page of M3TRICITY 2.0

3.2.4 Repository Analysis

The analysis of a repository starts with the user input of a URL and, as an optional parameter, the database type. The frontend contacts the backend through the public REST API endpoints that initiate the analysis with the `M3ConstructStarter` service of the History-Builder module. This service iterates through all files of each snapshot of the given repository in chronological order. The class `FilePathUtils` parses the path and extracts the file name.

Identification of the Entity Type

The first problem we are presented with is the identification of the entity type of each file. Each file can be categorized as one of the three different individuals of our model. Data files are structured or semi-structured files that contain data. This type is decided based on the extension of the file, which has to match `json` or `xml`. Binary files are identified based on the prefix of the file name. If it starts with a dot, the file is hidden on Linux and macOS, suggesting that it should be ignored. Filenames that contain zero or more than one dot indicate that the file does not contain source code. The last condition is based on the suffix of the filename. If it matches a long list containing common extensions of images, audio, video, archive, configuration, text, and more, it is not considered relevant for the application. If the file is neither data nor binary, it is considered as a type class.

Extraction of Metrics

The analysis continues with the creation of Java objects that represent each file as an entity. The types of entities share the information of the commits that introduced them, which packages they belong to, and a set of tags created with the path. Each individual needs a set of metrics that can faithfully represent it. The sub-module **Metric Extractor** does this.

We created a common structure using polymorphism for each entity type.

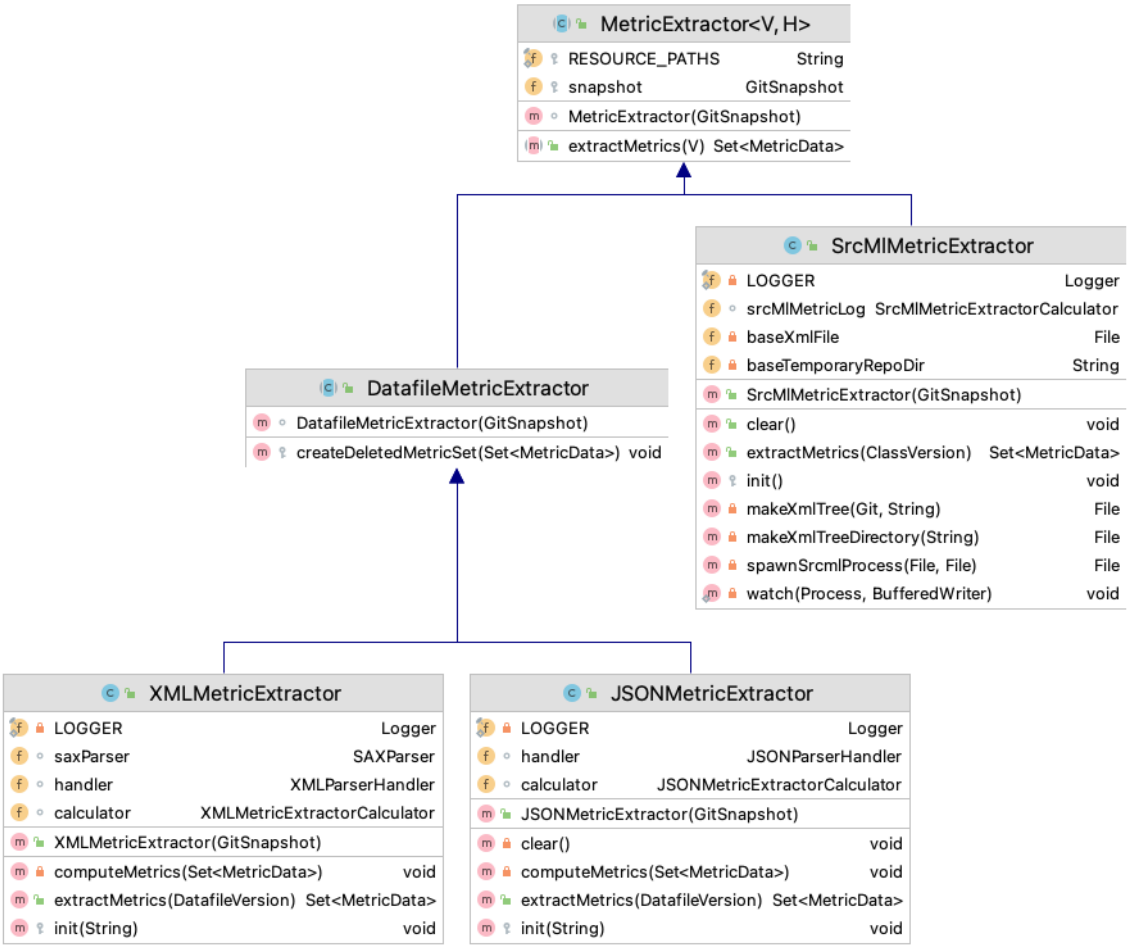
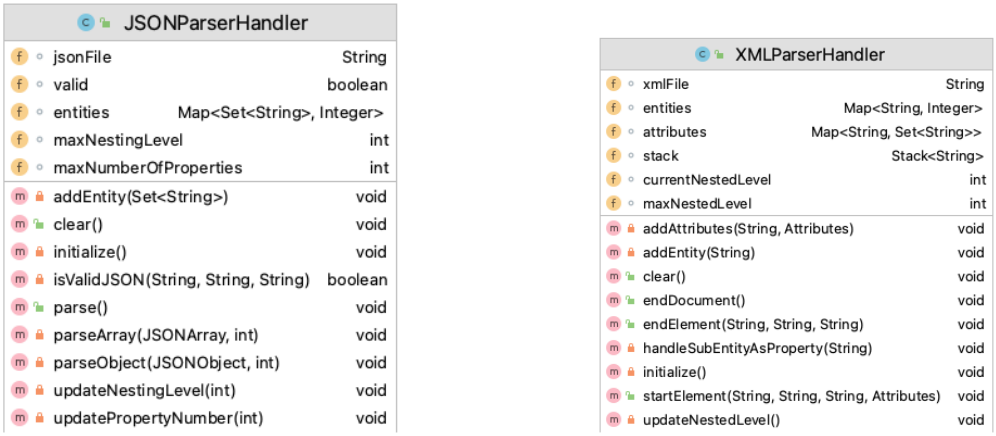


FIGURE 3.18: Class diagram of `MetricExtractor`

The abstract class `MetricExtractor<V, H>` is the superclass of the metric extractors. Figure 3.18 shows its hierarchy. The generic types `T` and `V` are, respectively, the class representing the version and the history of the entity. The abstract method `extractMetrics` initiates the extraction, and returns the set of metrics. `SrcMlMetricExtractor` is used for the type class. This class uses srcML[9] which creates an XML representing the class. Different structures of the data files require distinct logic to parse and extract. We created the superclass `DatafileMetricExtractor` and two sub-classes for each type. The **Parser** module parses the files. It consists of one class for each type, shown in Figure 3.19.

The class `JSONParserHandler` receives as input a path of a **JSON** file. This format allows the storage of a single object or an array of objects. We used a lightweight library, named json[10], to parse each part of the file into a Java object. For the analysis, each type of JSON object is identified by its key set. We have a map, named `entities`, of object keys mapped to the value of their occurrences. The maximum number of keys that an object has is stored in a variable and updated during the parsing of elements with more properties. We then check the value of each field and, if we find a JSON object, we recursively call the method with an increased level. The value of the maximum nested level is being checked and updated when analyzing a nested element. In the case of arrays, we analyze each element as explained above.

[9] https://www.srcml.org
[10] https://www.json.org



(A) JSON (B) XML

FIGURE 3.19: Class diagrams of the data file parsers

Regarding **XML**, there are different types of parsers. Document Object Model, also known as DOM[11], reads the file and creates a tree structure. This technique is expensive performance-wise. Simple API for XML, also known as SAX[12], iterates through all the elements, creating events at the start and end of each tag. It performs better for our use case scenario. The `XMLParserHandler` class uses this type of parser, implementing the `DefaultHandler`[13] class. The analysis starts with the opening of the root tag, continues by iterating through the file in order, ends with the closing of the root tag. We consider the properties to be attributes and sub-elements. We use a map named `entities` to save the name of the entities and their counters and a helper map `attributes` that stores the name of the entity and its set of properties. We use a stack to keep track of the nesting of the entity currently being analyzed. With the event of an opening tag, we count the entity found, save its attributes as properties, add its name to the parent entity properties set, and update the counter of nestedness. With the closing of a tag, we remove it from the stack and decrease the current nested level. After parsing the file, we clean the maps by removing the tags found only at the last level of nesting to count them as properties.

The second hierarchy has its root at the abstract class `MetricExtractorCalculator` and extracts the values of the metrics from the data structure created in the previous step. Figure 3.20 shows its structure. `SrcMlMetricExtractorCalculator` queries the XML created to retrieve the *Number of Methods*, *Number of Instance Variables* and *Number of For loops*. The value of *Lines of code* is created by counting the lines of the file with a `Scanner`[14]. Data files have slightly different data structures implemented in their metric extractors, thus needing separate calculator classes. `DatafileMetricExtractorCalculator` defines the methods. The value of *Number of Entity Types* and *Number of Entities* are extracted from the entities map by, respectively, its size and the sum of the values. The *Number of Properties per Entity* of a JSON is stored in a variable. For XML, it is the highest value of the attributes map. The *Maximum Level of Nested Entities*, in both cases, is assigned to a variable.

The complete set of metric values is returned and assigned to the entity then the analysis of the repository continues with the next file.

[11] https://docs.oracle.com/en/java/javase/11/docs/api/java.xml/javax/xml/parsers/DocumentBuilder.html
[12] https://docs.oracle.com/en/java/javase/11/docs/api/java.xml/javax/xml/parsers/SAXParser.html
[13] https://docs.oracle.com/en/java/javase/11/docs/api/java.xml/org/xml/sax/helpers/DefaultHandler.html
[14] https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Scanner.html

FIGURE 3.20: Class diagram of `MetricExtractorCalculator`

Creation of Databases, Tables and Accesses

The analysis of the database starts after the creation of a Java object representing the version of all files and packages of the snapshot. The **SQLInspect** sub-module analyzes the source code and creates the objects representing databases, tables, and table accesses. SQLInspect [36][15] is a static analyzer that inspects database usage in Java applications. This tool takes as input the source code, looks for the database creation statements, and infers the schema of databases. It supports multiple SQL dialects and MongoDB as a NoSQL database. We created classes to contact the command-line tool, with `SQLInspectHandler` as a superclass. The structure is shown in Figure 3.21. The database dialect is an extra input that can be chosen in the frontend. The analysis starts with the building of the process to run the tool. Each database type requires different parameters to be passed as a shell command. The `SQLInspectMongoDB` and `SQLInspectSQL` subclasses are used, respectively, for MongoDB and SQL. The relational database class allows the specification of the dialect. In case the user chooses not to have a database, `SQLInspectNone` skips the analysis and no database entity will be present in the repository object. After the input arguments are set, the process is created and SQLInspect is executed. SQLInspect saves the output of the analysis in external files, with slightly different structure of SQL and NoSQL databases which reflect the properties of the technologies.

The parsing of the analysis result is handled by the subclasses of `SQLInspectVersion`, which represents a single snapshot. `SQLInspectVersionMongoDB` and `SQLInspectVersionSQL` define the difference in the structure of the database types and the output files. Figure 3.22 shows the class diagram. It is divided into two parts: databases and accesses.

Unlike the entities that have origin in the versioning system, we do not have any knowledge about the changes made before comparing the outputs. We have helper data structures that save the last version of

[15] https://bitbucket.org/csnagy/sqlinspect

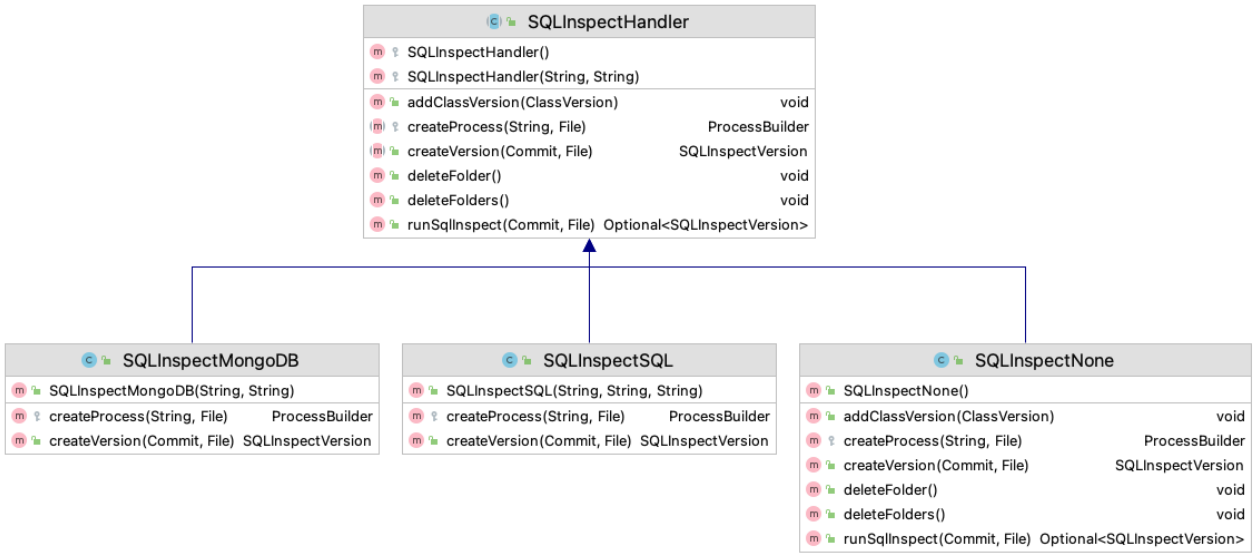


FIGURE 3.21: Class diagram of `SQLInspectHandler`

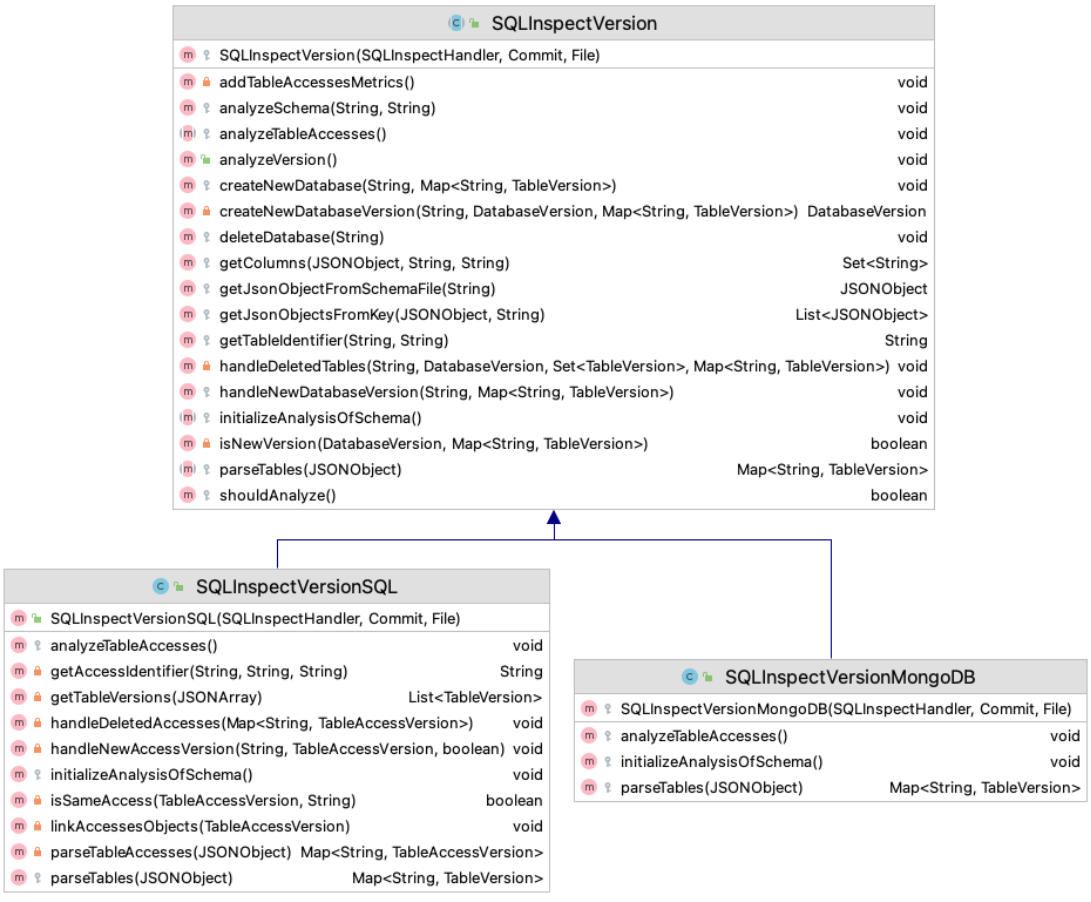


FIGURE 3.22: Class diagram of `SQLInspectVersion`

each type of object. We start the analysis by parsing the schema file into Java objects. The database contains a set of table objects. The table contains the list of columns found by the static analyzer. The metric *Number of Columns* counts them. The object is created with change type `ADD`. We then check if a previous version of such objects already exists, mark them with change type `MODIFY` and updated the helper data structures. The deletion of tables and databases is not trivial. We compare the names of the existing entities of the previous snapshot with current and create a new version with change type `DELETE`.

With the creation of Java objects, we can focus on the relation describing the interaction, stored in a separate file. For this part, we need an extra helper data structure to retrieve the class objects. We parse the file, extract the two parties of the relation, and represent it with a `TableAccess` object. It contains the `ClassVersion` with the query that accesses the `TableVersion`. At the end of the creation of the objects, we iterate through the tables and add the *Access*.

At the end of the execution of the module SQLInspect, the new entities are added to the object representing the version of the repository, and the files created by SQLInspect are deleted.

Histories of Entities

Each entity of each snapshot is represented as a version object. The history of each is created with the linkers. We refactored and extended this sub-module to support more types of entities. Figure 3.23 shows the structure of the classes. We created the `EntityHistoryLinker` abstract class that handles the creation of the history of an entity type. The subclasses are needed to create a new instance of the history object, which cannot be created in Java with generic types. The execution of the code is trivial. A version of an entity is created and its linker creates its history. New versions are added to the storyline with version number incremented by one.

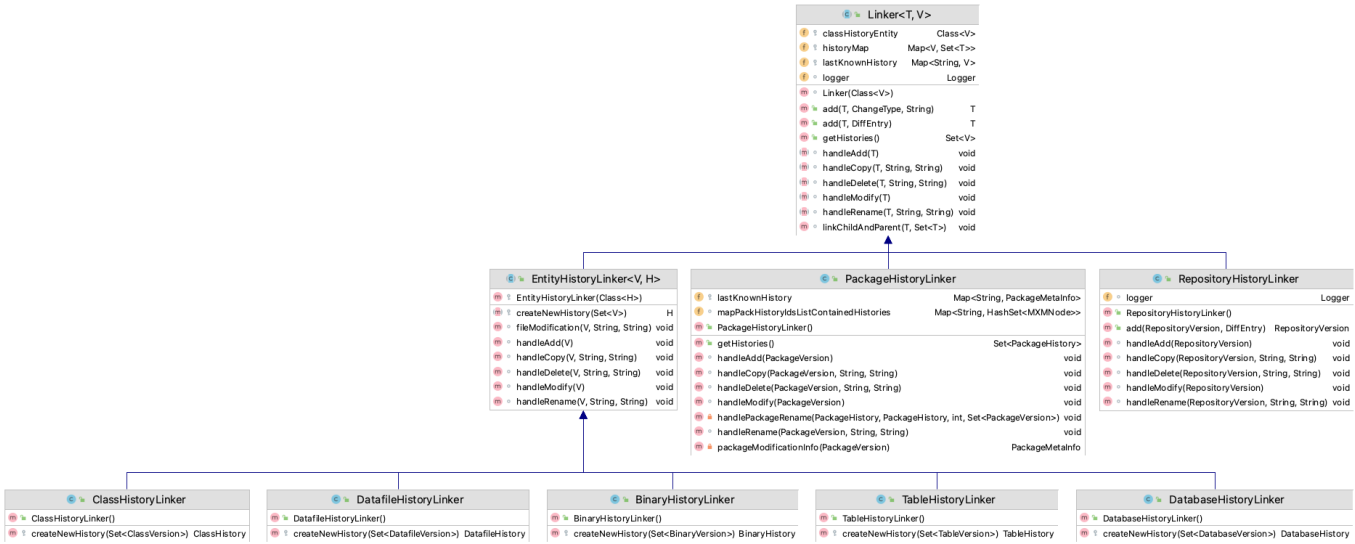


FIGURE 3.23: Class diagram of `Linker`

3.2.5 Database

At the end of the analysis of each snapshot, the `Repository` object represents the complete history of the project at the given URL. It contains the name and owner of the project, commits, the histories of all the entities and the table access relations. This object is ready to be stored. We decided to use MongoDB[16], a

[16] https://www.mongodb.com

document-orientated NoSQL database. It uses JSON-like documents with optional schema. We created a collection for each type of object.

Reference Problem

Our entities have many references, creating complex cycles. An object can be referenced several times by different instances. During the implementation, we encountered a problem with the database. We save each pointer to an object as a reference to it, which hampers the retrieval of data from the database. Each time a reference is found, the object referenced is loaded in memory with no check of previous loading or existence. This creates an infinite loop that can be broken only with memory exceptions or with the stopping of the execution. In the end, our `Repository` object is never completely loaded in memory.

Light Model

To prevent this problem we created a new *light* version of the model by replacing each reference with the ids of the referenced object. This, however, introduces complexity. We decided to keep the full model during the analysis of the repository. The rest of the application retrieves data from the database, thus using the new light version.

To preserve code readability, avoid unnecessary complexity, and reduce database usage, we created a wrapper class called `LightModelContainer`. Figure 3.24 shows its fields and methods. The idea of this

(A) Fields (B) Methods

FIGURE 3.24: Class diagram of `LightModelContainer`

class is to contain all the objects of a repository. This class is paired with a `RepositoryLight` object. Each type of object has a map of light object ids to the objects. At instantiation, the class contacts the database, loads all the objects related to the given repository name, and stores them in the correct map. This class exposes public getter methods for each object type, retrieves the object in the correct map, and returns it to the requested location. With the help of this class, we can load all the objects at once, thus avoiding extra time when retrieving the object directly from the database. It also assures that each object is loaded only once.

3.2.6 Visualization

The user starts the visualization by selecting a repository. The frontend contacts the backend through WebSockets that initiate the computation of the layout done by the View module. With the instantiation of a new view object, handled by **View**. The creation of the city layout is handled by the **Layout** component. These two components work together to create the most suitable city visualization of the repository. The history-resistant layout reserves a location for each each entity during the whole simulation. The user can personalize the visualization.

Layout Settings

Each entity type has a set of metrics, which can be chosen as the value of the size of the base, height or both. The values fall in the interval $[0, \infty)$. Classes are likely to have metrics larger than one; however, in rare cases, a metric can have a zero value. For example, let us consider a class that is used for storing values. Without logic, its number of for loops metric is 0. Also, an XML file with only one entity has a nesting level equal to 0. If it has no attributes, the value of the properties per entity metric is equal to 0. Regarding tables, it is impossible to have zero columns, as they are created by a query, but they can have zero table accesses if the tool did not find the class that accesses it. The corresponding mesh in the visualization cannot be rendered, which prevents its visualization.

We introduced a minimum size for each dimension, to avoid this problem. The length of a mesh's side is computed with the following formula:

$$a + b \cdot x$$

where a is the minimum value, b is the multiplication factor, and x is the value of the metric. The entities can be very small or very big, and the multiplication factor allows the user to adjust the base and height of each mesh type. The distance among buildings can also be changed, moving them closer or further to/from each other. The height of the districts can also be changed. The last setting available is the location of the databases with respect to the file system, which we explain next together with the computation of the layout.

Computation of the Layout

To have each mesh positioned at the same location throughout each version, we need knowledge about the histories of all entities. The first step is *learning* the structure of the repository. We use the two subclasses of `BinData`, shown in Figure 3.25, to map the container entities as building blocks. This is part of the **Bin** sub-module. Its id field matches the id of the package or database that it represents. The sizes map stores the dimensions of the buildings, mapping them to their id. `FileSystemBinData` represents a package, that can contain the three entities classes, data files, and binaries. Each of them has a map of the id of the history to the version that we want to represent. `DatabaseBinData` does the same for databases. Consequently, it has a map for the tables. The logic of this part is straightforward. We start by analyzing the histories of packages. For each version of a package, we iterate through the contained entities and save the highest value of the metric for the base of the entity in a `FileSystemBinData` object. During this part, we save the value of the highest element in the city. The peculiarity of this entity is its property to contain nested packages. We handle this by recursively calling the method to compute it, find the last nested package, and build it incrementally by representing a package as a class. We then use the same logic to build the `DatabaseBinData` representing the databases and their tables.

In the second phase, we need to add an extra layer of abstraction from the code and moving toward a city structure. We use the `Bin` class, shown in Figure 3.26. The idea is to remove the concept of the entity type and represent everything as blocks. The id of a Bin object matches the id of the history of the container, which can be a database or package. The dimension field represents the base of the container mesh. The `placedPieces` map connects entities' histories to their representing blocks. This object is constructed by the

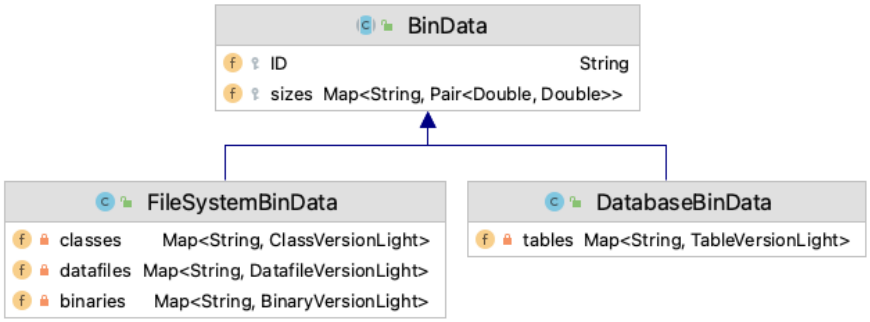


FIGURE 3.25: Class diagram of `BinData`

Packing sub-module. Given a `BinData` object, it loads the sizes of its entities, and places each by creating a `Block` object.

FIGURE 3.26: Class diagram of `Bin`

The third step takes care of the positioning of the two components of the city. The file system is represented as a city of buildings growing toward the sky. The databases are represented as platforms, with table cylinders growing toward the ground. Both parts have the same center position concerning x and y coordinates. We implement two visualization types as follows:

For the **City with Clouds** or **Cloudy City**, the file system is positioned at the ground level. The database platforms are positioned above the city, facing each other. We use the values of the highest building and table, found in the learning phase, to decide how much space to give to the city, and use it for the positioning of the databases.

For the **City with Underground**, the file system city is lifted, giving space to the databases and tables which are represented below. The value of the highest table, in this case, is used to determine how tall the underground is, and it is used for the offset of the entities.

Creating meshes information

Meshes can be created after the computation of the layout is complete. The logic of this part is as follows. First, we search the version of the repository that we want to represent. For each part of the city, we have a data structure that maps the id of the entity history to its `Bin` object representation, created during the computation of the layout.

The classes that represent the meshes in the backend, shown in Figure 3.27, are the following. The superclass `Mesh` describes the common fields of all meshes.

- `id`: the history id; it is used to identify the entity regarding the version.
- `repositoryVersion`: the value of the version of the repository; it can differ from the entity version.
- `authorString`: represents the email of the author of the commit and, therefore, author of the entity.
- `placeholder`: true if the mesh is a placeholder; used to show the existence of a container entity that does not exist in the current version.

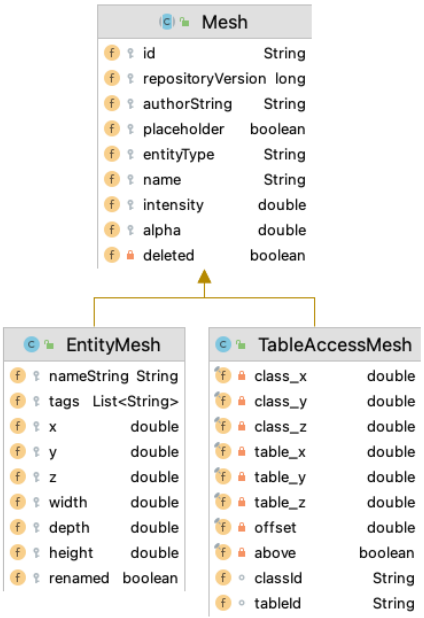


FIGURE 3.27: Class diagram of `Mesh`

- `name`: the name of the entity. For a file, it is its path. For a database or a table, it is its name.
- `intensity`: the value of the intensity of a mesh's color.
- `alpha`: the value of the transparency of the mesh. It is used for placeholders and database-related entities.
- `deleted`: represents if the entity has been deleted in the version and, thus, should disappear in the visualization.

`EntityMesh` represents packages, classes, data files, binaries, databases, and tables. They have nine extra fields. The id of the version is called `nameString`. The `tags` field is the list of tags of the entity. The position in the city is stored in variables x, y, and z, and their size is in variables width, depth, and height. The renamed boolean is true if the entity has been renamed, and therefore moved in the simulation. `TableAccessMesh` represents a table access of a class. It has the x, y, and z positions of the class and table that are part of the relationship and their history ids. The field offset is the value between the distance of the two parts of the city. The boolean above is true if the database platforms are above the city, otherwise, it is false.

In the implementation, we start by iterating the packages. We create a district mesh for each. We then iterate the modified classes, data files and binaries present the version of the package and create their meshes. We continue with the databases and proceed with the same logic for the tables. The location of the two parts of the city is handled by the two types of layout classes. The third step is the visualization of the interactions. The cloudy city connects the top of a class building with the bottom of a table cylinder. The city with underground is not trivial. The two parts of the simulation grow in different directions and face each other's basement. We solve this problem by adding an extra mesh for the class building below the surface. We then connect the bottom of that basement class, with the location of the table in the database. The meshes of the city represent the change of the version, which can mean that the class and table meshes can be missing if they have not been modified. We prevent this by saving the classes and tables that have interactions, to retrieve them when needed.

When this part is complete, the meshes are stored in a map with their repository-related data, and sent to the frontend through WebSockets.

3D Rendering

When a user asks for the visualization of a repository, the frontend asks the backend and waits for the response. The next step is the 3D rendering of the meshes.

The interfaces `MeshConfiguration` and `AccessConfiguration` represent the information of the mesh subclasses described in the previous part. The `M3Entity` class represents an entity of the city and is the superclass of each. The `CustomMesh` interface extends the `Mesh`[17] class of Babylon.js[18] real-time 3D engine library and is the superclass of each mesh of our visualization. We describe the implementation of the entity types and their mesh representations.

The `Cuboid` class represents a box mesh.[19] It is used to render a class, package, or database. The main differences between these entities are their color and transparency. `M3Class` uses a pure blue material, with 1 as blue and 0 for both red and green. The mesh representation in the city is never transparent, but the basement mesh below the city has light transparency. `M3Package` uses a gray material, with 0.5 for red, 0.6 for green and 0.7 for blue. If the package does not exist, it is visualized with 0.1 transparency. If it exists, it is fully visible. `M3Database` uses the same gray material of the package but with 0.8 transparency to show its inferred nature.

The `Cylinder` class represents a cylinder mesh.[20] It is used to render data files and tables, with different colors and transparency values. `M3Datafile` uses an orange material, with 1 for red value, 0.5 for green and 0 for blue. This color does not share components with the color of the class mesh, which helps their distinction in the visualization. The mesh representation is always complete and never transparent. `M3Table` uses a pure red material, with 1 for red and 0 for both blue and green. The tables have 0.8 transparency to show their non-existence in the parsed repository.

The `HemiSphere` class represents a hemisphere mesh.[21] It is used to render binaries by the class `M3Binary`. The red, green and blue values of the material are equal to 0.8 and the mesh is never transparent.

The table accesses are represented as lines[22] between the classes and the tables. The color is a gradient of the two linked meshes with opposite colors to hint the entity type on the other side.

3.2.7 General Improvements

Our work started with the understanding of M3TRICITY, followed by its improvement.

This tool lacked documentation, which slowed the comprehension process. While reading, we started its improvement by renaming variables, methods, and classes to a name that better described them. We aimed to improve readability, a key factor for the developers that are working on the project. We improved the design by introducing class constructors, object containers, and logic optimization, where needed. Part of the logic was over-engineered, and many blocks of source code were repeated throughout the codebase. We identified god classes with many fields, extracted them, and split them into new classes. Our refactoring aimed at reducing the complexity, optimizing, and aid further developers in a fast understanding.

[17] https://doc.babylonjs.com/typedoc/classes/babylon.mesh
[18] https://www.babylonjs.com
[19] https://doc.babylonjs.com/divingDeeper/mesh/creation/set/box
[20] https://doc.babylonjs.com/divingDeeper/mesh/creation/set/cylinder
[21] https://doc.babylonjs.com/divingDeeper/mesh/creation/set/sphere
[22] https://doc.babylonjs.com/typedoc/classes/babylon.linesmesh

Extensibility

Extensibility was not a primary design goal of M3TRICITY. Before starting our work, we changed and adapted parts of the project to support additional entities, metric extractors, different types of visualizations, and meshes. In the backend, we added hierarchies of classes and then added new subclasses for our project.

We modified the structure of the model to support new entity types. We moved fields located in an abstract class to the only subclass that uses them and moved common fields of the subclasses to their superclass.

The metric extractor was designed to be extensible to support more types of tools class-wise, which was not useful to us. We created two top superclasses to describe, respectively, the structure of a generic metric extractor and its calculator.

The view module is based on the idea of having a single visualization and two entity types rendered with the same mesh type. The sub-package glyph has a superclass and two subclasses with the same fields. We merged them into a single class. Our approach has different types of entities with distinct properties. We created an abstract class representing a generic mesh and two subclasses for the entities and the relations. In the inheritance of view, we extracted code blocks and created methods to be reusable by different entities. We moved the logic of the layout in a different sub-module to allow different visualization types.

The frontend handled each entity as a type class. The packages extended it by overriding the color used for the material. We changed the hierarchy at its root by moving the class representing a generic entity at the top of the structure. We then added subclasses to define the types of meshes and an extra level for the actual entities.

Design

The design of M3TRICITY had some flaws. We worked on its improvement design-wise, improving the logic and optimizing the tool.

This tool does not feature constructors as it creates empty objects and then fills the value of the fields using setter methods. With big objects, a new instantiation can take more than ten lines to be complete. We introduced constructors of different types.

The repository analysis was carried out before by a single method with 150 lines which called external methods of different sub-modules. A typical feature envy method. We created helper methods, in the same class, that do part of the analysis. The analysis uses various data structures. An example is `Linker`, which creates the histories. Each type of entity needs a separate instance object, creating many variables. We created `LinkersHandler`, a class that contains an instance of each type of entity linker.

The View module, with the help of smaller sub-modules, handled the creation of the visualization of the city. We refactored this module according to the template method design pattern and made the View classes abstract.

The class `EMVBinView` contains the settings of the visualization as class fields. We created new classes for the different settings and a new `Setting` class that contains all of them. This new sub-module is placed in the database core module. Part of the code repeated the same logic with slightly different settings. We created new methods, extracting the common parts. This module contained recurrent computations, or retrieval, of the same information. We removed this repetition by reusing the results. Some of this computation was based on finding one element of a list that satisfied a specific condition. We changed the data structure to use a map, which, on average, finds the requested element in $\mathcal{O}(1)$ instead of $\mathcal{O}(n)$, with n being the length of the list. During the creation of a mesh, if an error occurs, the methods return `null`. This was handled with checks of the value. We introduced `Optional` that either contains the mesh, if it was successfully created, or `empty` if an error has occurred.

The class `Packing` places the entities of a package into a container. M3TRICITY runs these services asynchronously, wrapping the result values inside of `CompletableFuture`[23] instances. On retrieval, the object needs to be checked for completion, before it can be unwrapped, and then a new instance of the object is created. However, if the computation is not complete, an exception is thrown, preventing the creation of the entity. M3TRICITY 2.0 runs the class synchronously, removing the unnecessary wrappers of the results. It removes extra steps in the computation and possible exceptions. The creation and update of the layout became significantly faster.

Visualization

Metrics are mapped to the sizes of the meshes' sides. Some metrics have a lower bound equal to 0, and related meshes were not visualized. M3TRICITY allows the user to choose a minimum size for the rendering. If the metric value is smaller than the minimum, then these two values are summed into a new number used for the side. If the value is equal or greater than the minimum, it is used without any addition. Buildings with smaller values than the minimum are rendered equal or bigger than the minimum building size. The files with metrics equal to the minimum can be visualized with the same size as the files with metric values equals to zero. It creates inconsistency in the visualization.

M3TRICITY 2.0 solves this problem by adding the minimum value to every metric, regarding its value. The visualization is consistent with respect to the software system. Each building of the city is rendered with the correct value. This aids the understanding of the system itself.

Database

Introducing a database in the project is a critical improvement. The tool can analyze repositories made of thousands of commits and files. The analysis can last days or even weeks. M3TRICITY stored objects in memory without the implementation of data persistence. It lost all data on shut down. This was a critical limitation. M3TRICITY 2.0 overcomes this limitation with a database as we described in Section 3.2.5. The implementation was not trivial, and we created a new version of the model to convert and store our Java objects in a real database. Our tool can be shut down and restarted without losing information.

Code Quality

We used SonarQube[24] to inspect the source code quality of M3TRICITY and M3TRICITY 2.0. Table 3.2 shows the comparison of their backends.

Project	Lines	Duplications	Code Smells	Bugs
M3TRICITY	6 567	7.3%	495	82
M3TRICITY 2.0	6 962	0.3%	17	0

TABLE 3.2: Comparison of the backend of the two tools

Our tool has about a thousand lines of code more than the original one due to the new features. However, we also cut several unused classes; therefore, the amount of new code is more than just the difference. We also reduced the duplication by extracting cloned code blocks into new methods. The code is more readable and easier to maintain.

M3TRICITY had almost 500 code smells, which is very high for a project of about 6000 lines. On average, every 13 lines suffer from bad practice. The most common and potentially harmful problem is the lack of parameters for generic types. We cleaned the code and removed them at each of our encounters.

[23] https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/CompletableFuture.html
[24] https://www.sonarqube.org

The number of bugs is also very high. The original tool has 82. The project contains 140 classes, thus, on average, more than half of them contain a bug. The most common and potentially harmful problem is the misuse of `Optional`[25]. This class allows having an *optional* wrapper of either a value or empty. A missing check for the existence of the value can cause a `NoSuchElementException`[26]. We fixed this type of bug by introducing checks for the content and handled the different cases. Other common harmful problems are caused by a missing check of the value of an object that can be null, causing `NullPointerException` that were also not handled and caught. Minor types of bugs are caused by using incorrect types of objects, without converting them. We fixed all the bugs and we did not introduce new ones. Hence we have 0 bugs in SonarQube.

[25] https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Optional.html
[26] https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/NoSuchElementException.html

Chapter 4

Stories of Cities

This chapter demonstrates M3TRICITY 2.0 in action. We analyze the evolution of information systems and describe them focusing on what could not be seen with previous approaches. We present screenshots for interesting cases. We start with JETUML, a lightweight desktop application for interactively creating and editing diagrams in Unified Modeling Language. We continue with the analysis of DP3T-APP-ANDROID-CH, the Decentralised Privacy-Preserving Proximity Tracing used by the SwissCovid app. We then describe GNUCASH-ANDROID, a mobile finance expense tracker application for Android. Finally, we show M3TRICITY 2.0. Table 4.1 shows the main size metrics of the most recent snapshots of the analyzed systems. The settings used for the visualizations of the cities are shown in Table 4.2.

Project	Snapshots	Lines	Classes	Data Files	Binaries	Tables
JETUML	2 044	26 391	269	37	195	0
DP3T-APP-ANDROID-CH	730	32 135	398	243	82	0
GNUCASH-ANDROID	1 730	51 871	348	289	631	11
M3TRICITY 2.0	373	13 505	180	4	40	21

TABLE 4.1: Statistics of the analyzed systems

Entity	Side	Metric	Minimum	Multiplicator
Class	Base	*Instance Variables*	2	1
	Height	*Number of Methods*	2	1
Data File	Base	*Number of Entity Types*	2	1
	Height	*Number of Entities*	2	1
Binary	Diameter	*Size*	5	0
Table	Base	*Number of Columns*	2	2
	Height	*Number of Columns*	2	2

TABLE 4.2: Settings used for the city visualizations

4.1 JetUML

JetUML[1] is a Java application to create and edit UML diagrams. The main branch of the project is composed of 2 044 commits of 19 contributors. Figure 4.1 shows the overall evolution, Table 4.3 shows the details of each snapshot.

Date	Time	Version	Classes	Data Files	Binaries
January 07, 2015	14:14	2	65	2	19
January 22, 2015	09:19	61	63	2	18
January 26, 2015	19:29	101	63	2	59
October 08, 2015	18:31	369	92	2	74
August 27, 2018	12:30	1 263	223	2	55
June 12, 2020	12:04	1 801	251	2	64
October 07, 2020	02:11	1 859	258	3	67
December 03, 2020	08:45	1 955	264	34	100

TABLE 4.3: Information of the snapshots of JetUML

On January 7, 2015, Martin Robillard created the project repository with a readme, a license and an ignore file. This first commit shows the added entities as binaries. This is important because M3TRICITY did not differentiate these entities from source files, but M3TRICITY 2.0 shows their true nature.

About a quarter of an hour later he initiates the project by creating **Violet**, and its sub-package framework, in the `com.horstmann` package. This contains about 70 Java classes and a few binaries representing properties and images. On the opposite side of the city, it is possible to see two data files. The first is tall and thin, it contains 287 entities of two different types. This snapshot is shown in Figure 4.1a.

This is followed by regular development tasks on Violet until, three days after the first commit, part of the classes are moved, creating **Violetta**, with the sub-package `graph` located under the `ca.mcgill.cs.stg` package.

Thirty versions later, on January 22, Violet and Violetta are merged into a package, giving birth to **JetUML**, in the new package `ca.mcgill.cs.stg`. The project has two test classes. They follow the same course of events in a new test package. This snapshot is shown in Figure 4.1b. The story continues with developers working on this new part.

Few days and thirty commits later, they introduce a new test and a folder named `testdata` containing binaries written in their notation with `jet` extension. This snapshot adds two classes with the same name in the source and test packages. The binaries are used by the tests of the new features. This is interesting, with previous approaches these files were seen as source files just like classes. Our approach can distinguish them and understand their relationship with the Java classes.

On the evening of January 26, they push a commit with the message *#34 Added the icons to the menus*. The **Icons** folder appears with 36 png images divided into four subfolders named according to the images' dimensions. This snapshot is shown in Figure 4.1c.

The development continues in the JetUML module. Their test-related packages and documentation are shown in Figure 4.1d. The developers create graphs and nodes then continue working on them for months.

They then start working on the graphical user interface. The **GUI** module has mostly Java classes and some binaries.

This is followed by a cleanup phase where buildings are either resized or disappear altogether. The icons also go through this process as they half in number. The next package needs refactoring. The complete cleanup phase lasts months and hundreds of snapshots.

[1] https://github.com/prmr/JetUML

In August 2018, three and a half years into development, the project is finalized for release. This snapshot is shown in Figure 4.1e. The city seems well-formed with a discrete size. The empty space is due to the modification and deletion of entities.

In the afternoon of February 6, they decide to work with the images, and the content of the `icons` folder is modified, filled, and deleted in a series of commits.

In June, the `Prototypes` class is added with a peculiar shape reminding of a parking lot. This class has 31 fields, 3 methods, 145 lines of code, and no loops. It looks like a data class. It can be seen in the bottom part of Figure 4.1f.

At the end of August 2020, they add one JSON file and three images with the `tip` name prefix. A few weeks later these files are moved into a specific folder, named `tip_resources`. This snapshot is shown in Figure 4.1g.

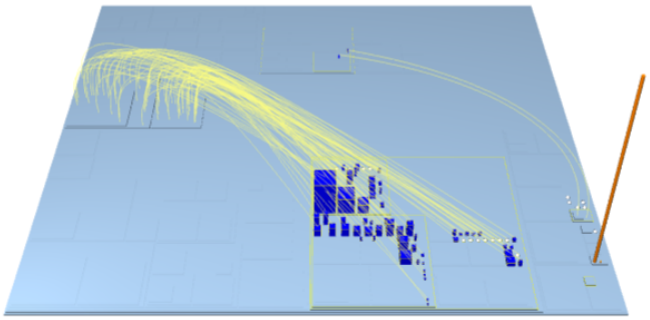
This event is followed by the creation of a test class for `tips`. It is born with a big base and short height, then grows in height, standing out in the city, then it stops growing and becomes less noticeable. The package of data soon disappears.

In October, a new folder is created named **tipdata** with the two sub-folders: `tips` for JSON files, and `tip_images` for images, with two entities each. Few months and 50 commits later, each folder has more than 30 entities. This snapshot is shown in Figure 4.1h.

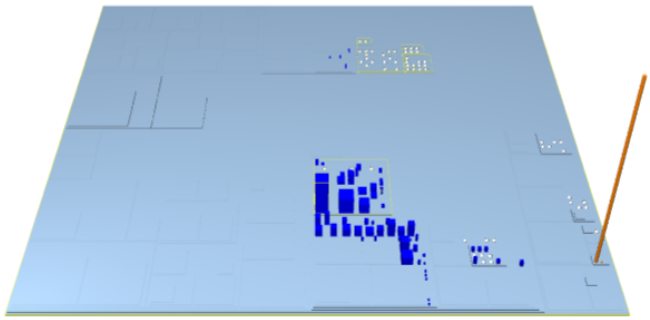
The repository continues its life with minor changes.

The city looks nicely organized. The left side contains the source code on the bottom, and the test on the top. It is interesting to notice that their sizes are similar, indicating that testing is an important part of this system. The right side is reserved for data files, binaries, and documentation. The sparsity of the source code reflects the creation, renaming and deletion of classes. The test package, on contrary, is more compact. The size of the classes change during the evolution but deletions are not common. The city empty space represents the folders that are not present anymore or have been moved.

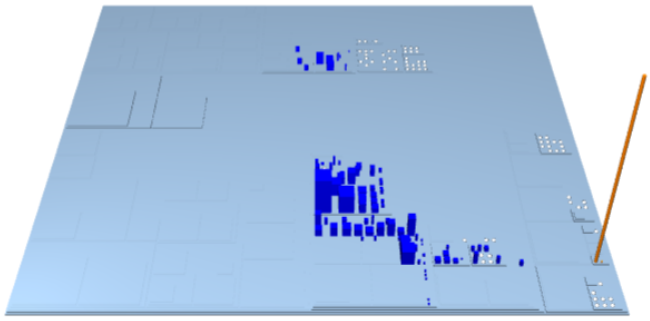
(A) Wednesday, 7th of January 2015 at 14:14



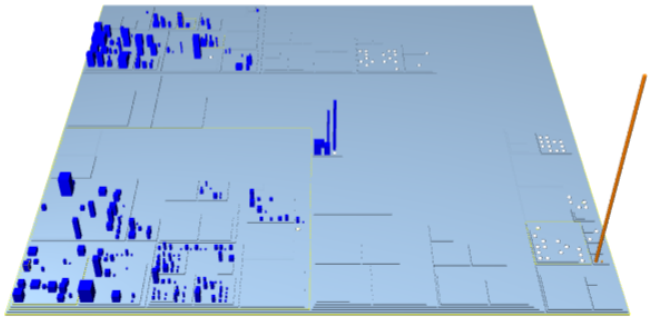
(B) Thursday, 22nd of January 2015 at 09:19



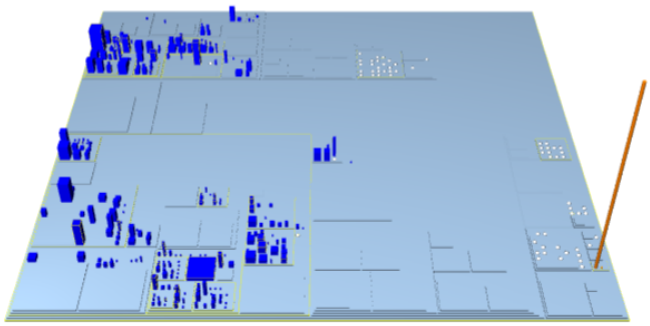
(C) Monday, 26th of January 2015 at 19:29



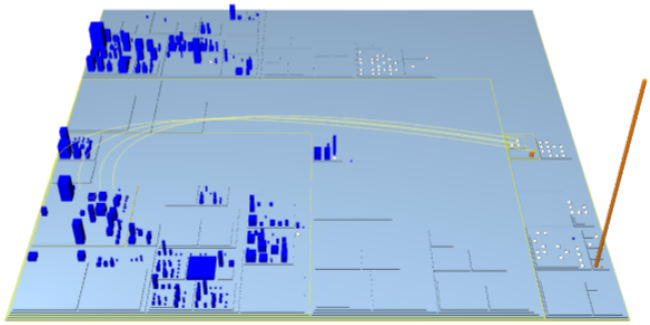
(D) Thursday, 8th of October 2015 18:31



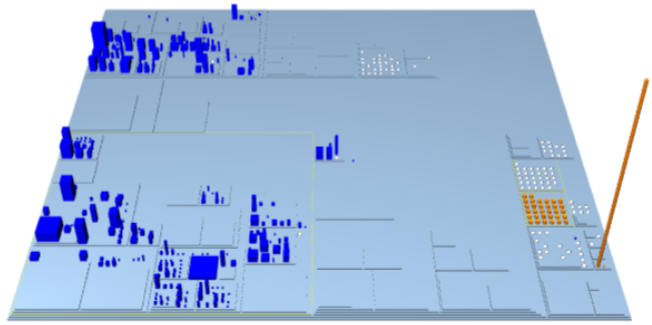
(E) Monday, 27th of August 2018 at 12:30



(F) Friday, 12th of June 2020 at 12:04



(G) Wednesday, 7th of October 2020 at 02:11



(H) Thursday, 3rd of December 2020 at 08:45

FIGURE 4.1: Evolution of JETUML

4.2 DP3T-APP-ANDROID-CH

DP3T-APP-ANDROID-CH[2] is the Decentralised Privacy-Preserving Proximity Tracing protocol that uses Bluetooth Low Energy for COVID-19 proximity tracing. This Android app is developed in Java with a small portion of HTML and a few Kotlin files. The main branch of the project is composed of 730 commits of 12 contributors. Figure 4.2 shows the overall evolution, Table 4.4 shows the details of each snapshot.

Date	Time	Version	Classes	Data Files	Binaries
May 02, 2020	13:55	2	107	154	48
May 04, 2020	10:26	3	107	154	48
June 03, 2020	16:27	146	118	159	62
September 22, 2020	10:48	421	168	179	68
March 10, 2021	11:26	658	195	222	68
April 29, 2021	10:45	730	198	230	68

TABLE 4.4: Information of the snapshots of DP3T-APP-ANDROID-CH

On April 15, 2020, Fabian Aggeler creates the project repository with a readme, a license and an ignore file. A few hours later Simon Rösch creates the initial version of the Swiss app by adding 308 Java classes, 154 data files, some HTML, and many binaries representing images and text. Most of the source code is placed in the `org.dpppt.android.app` package. This is the birth of the application. The city, shown in Figure 4.2a, is well composed. The lower part is reserved for the source code. The middle left is dedicated to the web part. There are several districts composed of data files dedicated to the **resources**. The cylinders are of different heights and widths. The following day, Christoph Maurhofer, changes the name of the main package to `ch.admin.bag.dp3t`, matching the name of the official Federal Office of Public Health of Switzerland. Figure 4.2b shows that the source code is moved to the left with the restructuring.

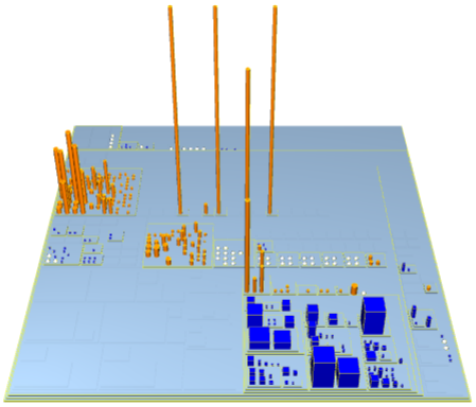
The **Values** module is composed of folders with XML files, named `strings`, that store the text of the application in different languages. These files are often modified, growing in height but never in width. This means that data is added in the file, but it is always of the same type.

About two and a half months and 150 commits later, six new languages are supported by the app, resulting in six new districts with the respective data file. This snapshot is shown in Figure 4.2c. The app continues its life with some modifications in the source code, new information in the data files, and new binaries. On the top left, a new district dedicated to the documentation is born. A few days later, some images are moved to new folders, indicating a restructuring of the folder structures.

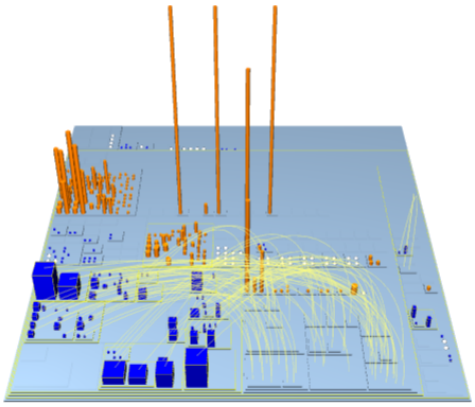
The app continues to evolve with minor changes in the size and shape of the source code buildings. There are few small ones, some noticeable big and a few "fat" but "short" classes. With the modification of source code, the data files grow in size. This hits for coupling: the classes modified are the ones that use the data stored in the cylinders. As time passes, the files become taller. On July 20, more languages are supported, adding new data files and images, representing the new parts of the world.

The developers continue the development. The class `SecureStorage` becomes bigger and taller with time, indicating that security is important. On the morning of September 9, after some weeks of inactivity, part of the system is moved by a package rename, shown in Figure 4.2d. On a Friday evening in the middle of December, so much information is injected into a JSON file (introduced two days earlier) that its height explodes. At the beginning of March 2021, the `SecureStorage` class finally looks like a real building. It continues growing both in width and height for the following month and then shrinks, suggesting that a refactoring has occurred. Figure 4.2e shows the bigger version of this class. It is the biggest entity in the city. Figure 4.2f shows the final version of the city. The city looks nicely organized. The bottom half is dedicated to the source code. The top part is used for the storage of data and binaries.

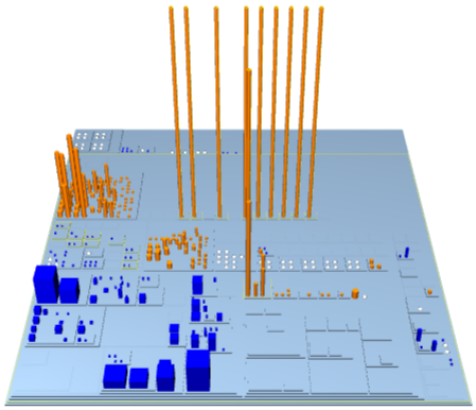
[2] https://github.com/DP-3T/dp3t-app-android-ch



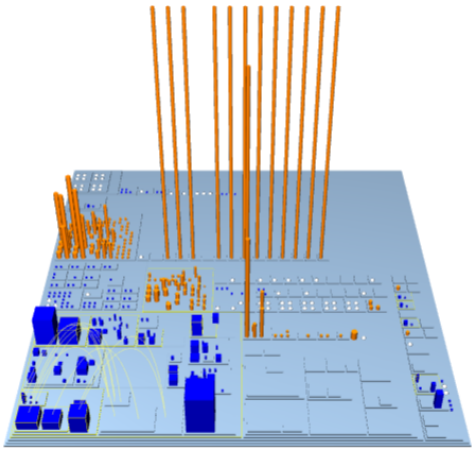
(A) Saturday, 2nd of May 2020 at 13:55



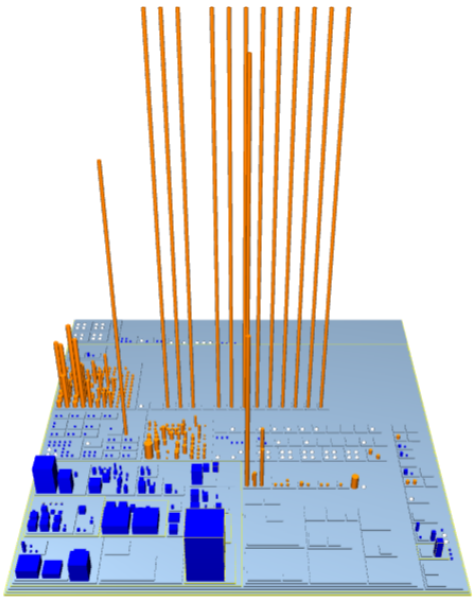
(B) Monday, 4th of May 2020 at 10:26



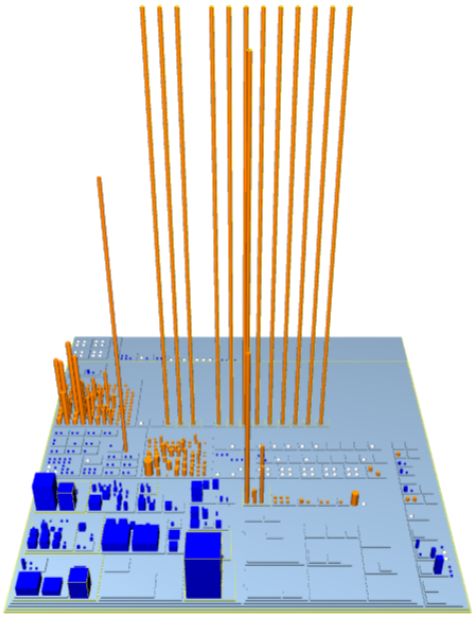
(C) Wednesday, 3rd of June 2020 at 16:27



(D) Wednesday, 22nd of September 2020 at 10:46



(E) Wednesday, 10th of March 2021 at 11:26



(F) Thursday, 29th of April 2021 at 10:45

FIGURE 4.2: Evolution of DP3T-APP-ANDROID-CH

4.3 GNUCASH-ANDROID

GNUCASH-ANDROID[3] is the Android companion app of the GnuCash accounting program. It allows to record transactions on-the-go to later import the data into GnuCash. The main branch of the project is composed of 1 730 commits of 46 contributors. Figure 4.3 shows the overall evolution, Table 4.5 shows the details of each snapshot.

Date	Time	Version	Classes	Data Files	Binaries	Tables	Accesses
May 24, 2012	19:25	3	90	86	242	2	2
November 04, 2012	17:20	108	123	140	395	2	3
January 31, 2013	00:29	181	39	53	136	2	0
September 18, 2015	19:06	1050	157	121	248	2	5
December 28, 2015	16:35	1250	188	152	271	3	4
December 02, 2020	08:13	1730	217	163	232	3	6

TABLE 4.5: Information of the snapshots of GNUCASH-ANDROID

On May 13, 2012, Ngewi Fet creates the project repository with 83 Java classes, 85 data files, and 243 binaries. The source code is mainly located in the `com_actionbarsherlock` package, nicely divided into sub-packages. The `res` folder contains three districts of images and several districts of data files. Separately, the module **GnucashMobile** has with one Java class, four data files, and some binaries. This is the birth of the application.

Ten days and two commits later, the database is introduced, with two tables paired with two Java classes named after them, and four containing the word database in their name. The class `DatabaseHelper` accesses both tables. The snapshot is shown in Figure 4.3a. This is important because M3TRICITY was not able to know if databases existed, could not visualize them or that the source code is interacting with them.

The system continues its growth, with few stale commits described with *fixing bug*. The tables grow in size as new columns are added. The buildings continue to grow slowly, some getting smaller due to some *refactoring*, while data files keep becoming higher. For example, the `string` files, containing the values of the text of different languages, have more than 400 entities each.

With time and more source code, test classes become bigger and new buildings are introduced. This means that the developers are testing the new features. With more commits, new languages are supported, adding the related data files containing the values of text to display. Binaries are also added, as support for the rest of the application. Figure 4.3b shows the city after these changes. There are more buildings of source code and they are noticeably bigger.

On December 12, the `TransactionsDbAdapter` class starts accessing table `transactions` that now now two accesses.

On January 31, the project goes through a big restructuring. Almost 450 files are deleted, and 220 are moved with the renaming the folder `GnucashMobile` to **app**. The database-related classes are still alive but the accesses to the tables are removed. This is shown in Figure 4.3c.

At the beginning of February, `DatabaseHelper` and `TransactionsDbAdapter` restart to access the database. A few days later, the class `AccountsDbAdapter` starts accessing table `accounts`. Now, each of the two tables has a private class that accesses them. In the following days the adapter class of the `account` table continues growing, doubling its original size. During this time, new tables are added to the database, that are soon deleted. This happens often. An example is the `split` table, added with its helper classes that accessed it, soon grows and then disappears.

New classes are introduced and, with time, few "parking lots" can be spotted. These classes have many variables but only a few methods.

[3] https://github.com/codinguser/gnucash-android

The correctness of the source code is checked with the creation of new tests. The database evolves with the addition and quick deletion of tables. This time it is `schedules_events`. The tests are then moved to a new package and soon another table disappears. Normally, the project has two to three tables and the temporary ones that exist in few snapshots. For example, table `account` is one of the first added, gets deleted, then brought to life and killed again.

Around this time, the developers start to work on the **User Interface** module. Figure 4.3d shows the status of the project in the middle of December 2015, three years after its creation.

The repository continues its life with minor changes, due to regular maintenance, and the usual short-lasting tables.

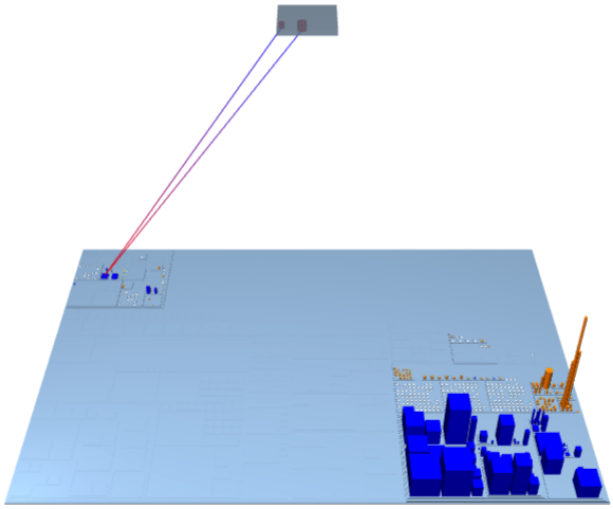
The developers work on tests, making them bigger and bigger, shown in Figure 4.3e. The source code buildings keep changing slightly in size, indicating standard maintenance.

A new district of data files is born with different characteristics. They are "fat" and "short", meaning that they have more types of data rather than much data of the same type. December 2, 2020, is the last snapshot of the system. It is shown in Figure 4.3f.

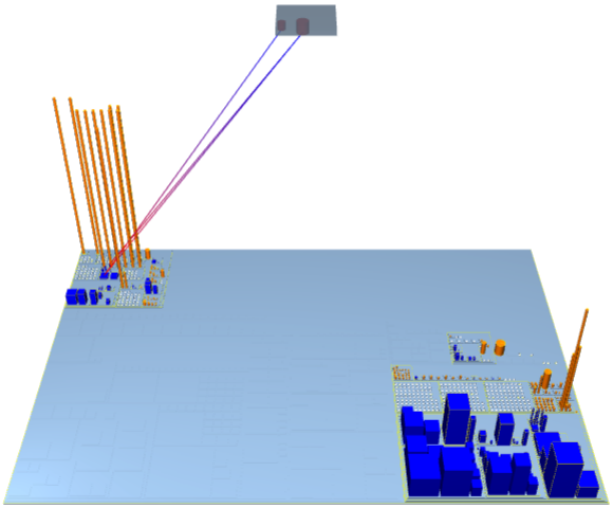
The city is most alive in the bottom left part. Most of the classes are small in size and with cubic shapes. A few are surprisingly wide, what we referred to as 'parking lots'. One class, named `DatabaseSchema`, does not have any methods. The other two have few floors, meaning that some methods are present. The size of the base is very wide. These classes have about a hundred instance variables.

The test package is in the city center, next to the district composed of harmonic data file cylinders. A smaller district can be seen at the bottom center. They are small in size.

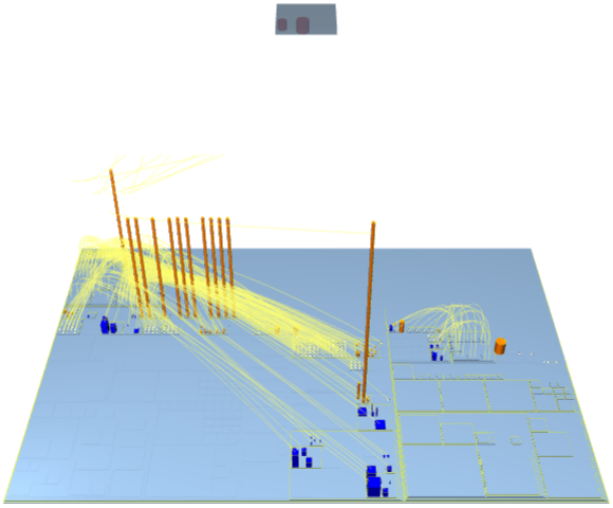
There are some districts of binaries throughout the city. The database is composed of three tables: `transactions`, `splits` and `scheduled_actions`. The first is the only table that lived during the whole database existence. The other two tables were added and deleted multiple times. Interesting to notice that the `MigrationHelper` class accesses all the tables of the database.



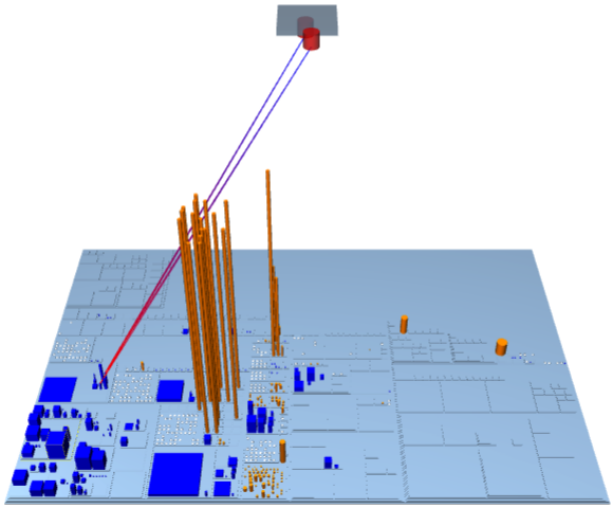
(A) Thursday, 24th of May 2012 at 19:25



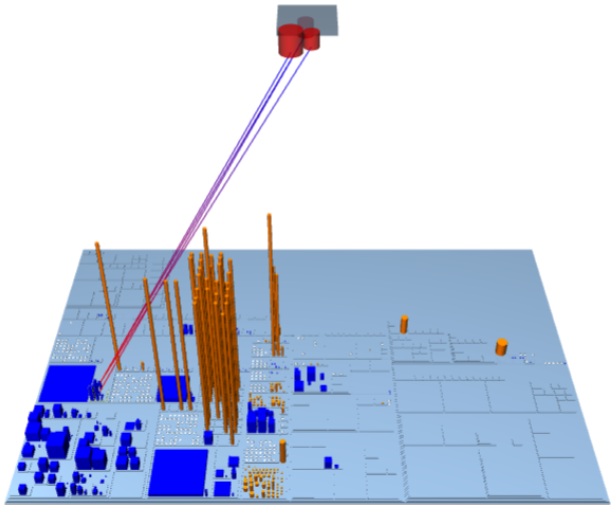
(B) Sunday, 4th of November 2012 at 17:20



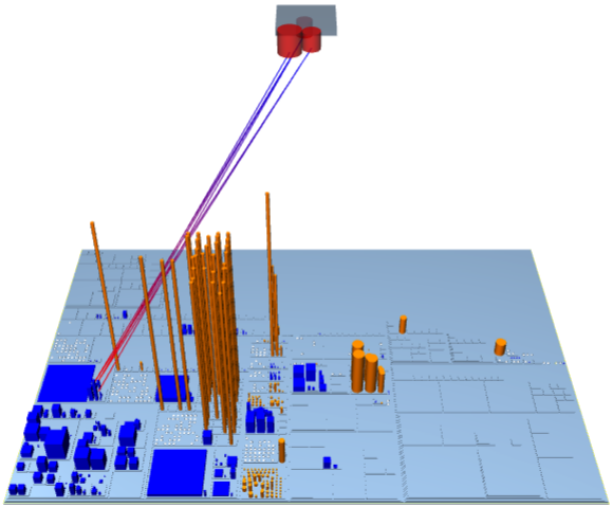
(C) Thursday, 31st of January 2013 at 00:29



(D) Friday, 18th of September 2015 at 19:06



(E) Monday, 28th of December 2015 at 16:35



(F) Wednesday, 2nd of December 2020 at 08:13

FIGURE 4.3: Evolution of GNUCASH-ANDROID

4.4 M3TRICITY 2.0

M3TRICITY 2.0 is our tool created for this thesis. It is interesting to use our project to analyzed our project. It is developed in Java and TypeScript. The main branch of the project is composed of 372 commits of 2 contributors who worked at different times, dividing the story into two parts. Figure 4.4 shows the overall evolution, Table 4.6 shows the details of each snapshot.

Date	Time	Version	Classes	Data Files	Binaries	Tables
April 11, 2020	10:50	2	56	0	1	0
August 13, 2020	09:23	193	261	3	40	0
March 09, 2021	11:06	228	292	2	34	0
March 18, 2021	13:55	250	268	2	32	0
April 03, 2021	11:18	264	271	2	32	0
April 05, 2021	17:13	270	304	3	32	9
April 07, 2021	20:35	277	277	3	32	9
April 08, 2021	21:49	283	286	3	29	12
April 18, 2021	16:24	304	284	3	29	16
June 01, 2021	16:54	372	299	3	33	16

TABLE 4.6: Information of the snapshots of M3TRICITY 2.0

On April 10, 2020, Federico Pfahler creates the repository with a readme file. The following day he adds the *city server*, which is the main backend composed of packages for the web communication and the database with the model, store classes, and services to retrieve the in-memory data. This snapshot is shown in Figure 4.4a. The districts are located in the right bottom part of the city. The following Monday afternoon the frontend is introduced in the project, creating a new central district on the opposite side of the city. The city continues its construction with new buildings in the backend, coupled with their visualizations in the frontend.

In the middle of May, about a hundred snapshots later, Federico Pfahler works on the implementation of the city layout. He tries several algorithms, adds new classes, modifies, and then deletes them. The fourth version of **Packing** is the final one, which is still used today. The project continues its life with about 50 commits and new features. An updated version of the city is shown in Figure 4.4b. On the afternoon of September 8, Federico Pfahler makes his last modification. The project is able to visualize repositories as evolving cities.

On the afternoon of January 25, Susanna Ardigó starts working on the project. Her first change is the renaming of the main package to the name of the research group of the thesis. This moves the backend from the right to the left side of the city. A few weeks later, on a Tuesday evening, she begins the **improvement** of the source code with various refactoring and improving the project to be extensible, its new design goal.

About a week and ten commits later, new classes are added to the **Model** package to represent databases and tables. This is followed by bigger constructor and view classes, and additions in the frontend for their visualization. Figure 4.4c shows the status of the city after the modifications.

On the afternoon of March 9, one month into working on this project, **Data Files** are born. This type of file was previously considered as source code. Its introduction is reflected in a taller `M3Constructor` class and new files in the frontend.

Thursday, March 18, Susanna Ardigò decides that she wants a cleaner code. She removed unused code and classes, and refactored the main components. A new class representing a snapshot of git is added. A

new class is created to store the setting of the visualization, moving the information out of the view classes. This snapshot is shown in Figure 4.4d.

The focus moves to the metrics of the new entities. The **Parser** package appears with an XML and a JSON parser class followed by their metrics extractors. The sizes of the buildings are different until the beginning of April when the commit *'updated data file metric calculations to be consistent for different extensions'* is pushed. Figure 4.4e shows the updated city.

Two days later, on a Sunday afternoon, a commit begins a new era. A new version of the model is created and used to **store information in the database**. Its visualization appears with eight tables, and a new package dedicated for the preparation of the data featuring the class `DatabaseHelper`. Susanna Ardigò saved the database schema in a JSON file, located in the center of the city. This is shown in Figure 4.4f. The central district, placed in the middle left, becomes its complement. The model was previously located in the generic database package. The entities are moved in a new subfolder, while the new model is also placed in a new subfolder. This can be seen in the image. A new district dedicated to the database services is added. It is composed of small classes and a big building in charge of database communication. The removal of the old in-memory database and their packages resulted in thinner constructor and view classes, which replaced the instance variables of each entity type with one single instance to communicate with the database. This process lasts about a week. Figure 4.4g shows the result of these changes. The database packages have bigger buildings and an extra district for the data transfer objects, which were previously placed in the web folder, located in the bottom center that is now empty. In the bottom left the two big buildings became smaller and thinner.

On the evening of April 8, the project is introduced to **Binaries**, a new entity type. This creates new classes in various parts of the system, more tables, and a taller schema data file, shown in Figure 4.4h.

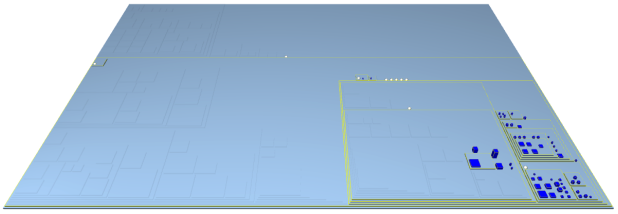
The following Monday, the focus is back on **Databases**. The new package `SQLInspect` is created, matching the name of the tool used to analyze database interactions. This lasts five snapshots and a week.

On April 18, the visualization of the city becomes more customizable with the addition of settings to trim each side of each entity type. Figure 4.4i shows the status of the city. This is reflected in the new **View Setting** package and slimmer view classes, due to the removal of the information previously stored as instance variables.

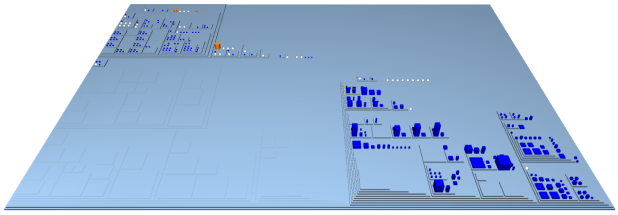
On April 26, new classes appear in the SQLInspect folder reflecting the new database types that the tool supports.

On May 1, the feature to delete a project adds height to `DatabaseHelper`, becoming noticeably tall. In the afternoon, the support of new types of layout is introduced, moving logic in a new package, making the view classes smaller. This ends with the addition of the last layout class, a city without the database.

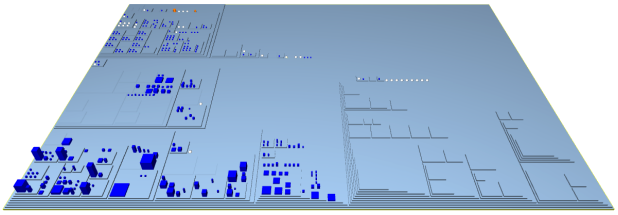
After few minor changes, Susanna Ardigó commits her last change on June 1st. Figure 4.4j presents the final snapshot of the city. The top left part shows the frontend. It is mainly composed of source code. It has some binaries and a few data files. The bottom half of the city is divided into two parts. On the left, there is the backend of M3TRICITY 2.0, on the right, the placeholder of the initial version of M3TRICITY. The new backend has a big package dedicated to the model. It holds two different versions, the database services, and the data transfer objects that were previously in the web package. The bottom part is divided into three parts: the view on the left, the analysis in the middle, and the web on the right. The city looks nicely organized.



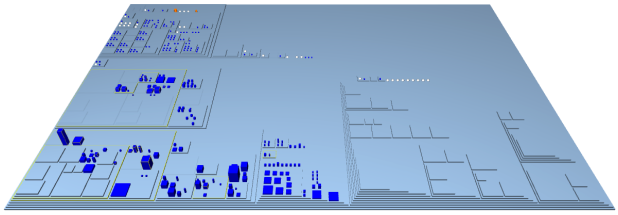
(A) Saturday, 11th of April 2020 at 10:50



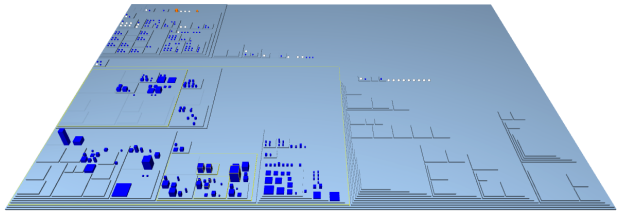
(B) Thursday, 13th of August 2020 at 09:23



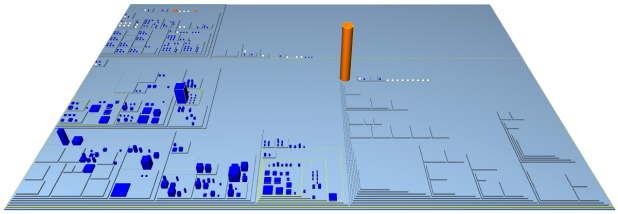
(C) Tuesday, 9th of March 2021 at 11:06



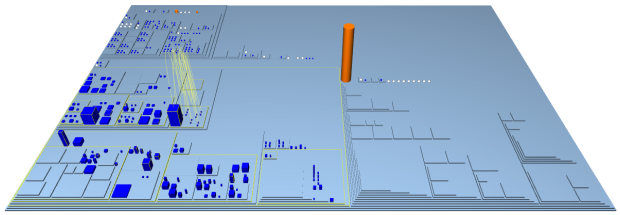
(D) Thursday, 18th of March 2021 at 13:55



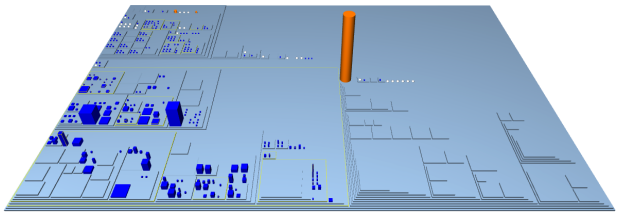
(E) Saturday, 3rd of April 2021 at 11:18



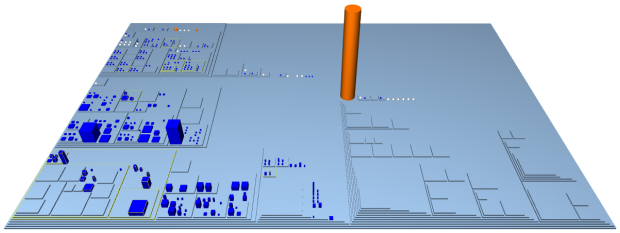
(F) Monday, 5th of April 2021 at 17:13



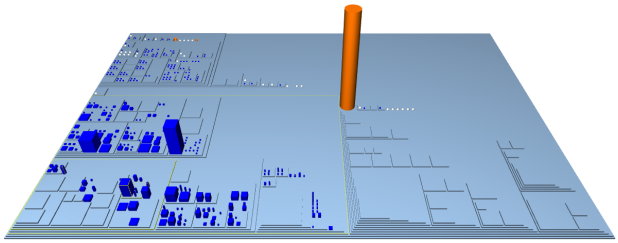
(G) Wednesday, 7th of April 2021 at 20:35



(H) Thursday, 8th of April 2021 at 21:49



(I) Sunday, 18th of April 2021 at 16:24



(J) Tuesday, 1st of June 2021 at 16:54

FIGURE 4.4: Evolution of M3TRICITY

4.5 Summary

To validate our approach to visualize information systems, we used M3TRICITY 2.0 to analyze four projects. We chose an application with data files and binaries and, a data-intensive application without any database, an Android appwith a relational database, and our web application with a NoSQL database.

The first application is JETUML, a lightweight desktop application to create UML diagrams. The analysis showed part of the repository that was hidden in previous approaches. This system is mostly composed of source code, with a few data files and binaries. From this system, we learned that source code is accompanied by binary files. The creation of this application required time. Its maintenance did not introduce any major change, but it can be followed by the addition of new features. In this case, toward the end, data files appeared. This shows that the new storage types can be introduced, even after the deployment.

Second, we analyzed DP3T-APP-ANDROID-CH, the official Swiss Coronavirus tracing app. The analysis was able to reveal the numerous data files usually used by Android apps. Various districts of the city were dedicated to data. The values of the text of different languages stored in XML showed their content stands out in the city. The visualization showed that this Android app is rich in data files. Most of them contain much data of few entity types. The source code reached a stable point soon during its lifetime. Then the system continued its life mostly with growing data files.

Thirdly we analyzed GNUCASH-ANDROID, an Android app to save money transitions on the go. We chose a system centered with few data files and binaries and without any database. The visualization of this system was interesting due to the existence of a database and the accesses made by classes. This repository, like the previous one, is rich in data files, most of them have much data of few entity types. This system was re-engineered several times. From this system, we learned that, due to having a database, part of the source code is dedicated to communication. During its lifetime, one to two classes were in charge of this task. Each table, though, also had its class for the database communication. Their shapes were peculiar. The table-specific classes were “tall” and “thin”. The generic database classes were “fat” and “short”. This suggests that the generic ones keep holding the information of the tables, while the table-specific classes have the methods to access the database.

In the end, we analyzed M3TRICITY 2.0 to verify the resulting visualization and compare it to our experience developing it. For this reason, this is the most consistent analysis. This system is rich in source code, with few binaries and almost no files. The evolution was slowly spread over time. From this system, we learned that the database is a central component. That is reflected during its life. At first, the system saved data in memory. Each entity had its class to save the instances. When a proper NoSQL database was introduced, the classes were replaced by tables that allow persistent storage. This required efficient communication, reflected in the creation of a package with this sole purpose. With the addition of a deletion feature, the class used for direct communication was noticeably big. This system was rich in classes, widely spread throughout the city. A very interesting aspect about this system is its size. M3TRICITY and M3TRICITY 2.0 share a common part; still, they are very different. The new system is more complex, yet it is not much bigger than the first. This shows the benefits of refactoring and code-optimization.

Chapter 5

Conclusion

In this chapter, we summarize our work and state possible future work.

5.1 Summary

The main goal of this master thesis is to visualize the evolution of software systems, their data, and their interactions using the city metaphor. The two parties (repository and database) are of different nature. Having them in the same visualization, required a new representation. We decided to visualize the new part above or below, allowing the creation of a direct mapping for their interactions. We explored different entities in a repository, creating different visualizations for the user to easily distinguish them.

With M3TRICITY 2.0, the web application that we developed for this thesis, we augmented the current visualization of the city, adding new shapes inside and outside of it. We created different types of entities, representing different file types. Each has its own set of metrics, shape, and a particular color. We constructed new entities that are not physically versioned in the repository. Our solution can handle multiple types of databases and dialects. We added a new part of the city with two different visualization: city with clouds and city with underground. These new parts are see-trough, to show that they are inferred. We visualized the table accesses made by classes, showing a link in the visualization. This relation goes beyond our original expectations, linking the two parts of the city.

We then used M3TRICITY 2.0 to analyze open-source projects. The new analysis revealed interesting observations about the evolution of these systems. We can see that the same types of systems tend to have similar patterns. For example, Android apps share the peculiarity of storing text in XML files, creating a different district for language.

Overall, M3TRICITY 2.0 was fascinating because it shows software and data, which are the two main components of modern systems.

5.2 Future Work

Visualizing software and data together in the same city opens new possibilities. We explored new visualization, that gave us possible future works.

5.2.1 New Entities

Our approach focused on visualizing source code and data. We also added binary files which are of different nature. They can be images, videos, text, configuration, and more. For example, an ignore file as `.gitignore` is useful because it filters what should and should not be versioned. Images and videos can be shown in the user interface. The `README.md` file should always be in a repository with the description of the project and how to use it. Their use structure and use are very different, but their relevance application-wise is very similar. We decided to represent them the same way. They are not relevant to the application. New approaches can study their nature, analyze them and create new visualizations.

5.2.2 Different Visualizations

The container entities, Packages and Databases, do not have metrics. This is due to the fact that metrics are created to used them for mapping values on the size of the meshes. These entities are created as containers, based on the meshes that they contain. New metrics can be added for the creation of new visualizations. A Package can have *Number of Classes* and *Number of Packages*, a Database can have *Number of Tables*. They can be used to represent their content. Packages, in addition, can also have *Nested Level* to represent the location in the hierarchy.

5.2.3 Live Systems

Projects can be deployed on the web and accessible to anyone with an internet connection. These systems are live and alive. Data can be retrieved continuously from the server, it can be analyzed and visualized. It would be interesting to look at a real system and see which parts of the code are currently being used at that very moment. This can be done by highlighting the currently running parts of the city. The visualization of the table accesses can have more details, reflecting what is happening. For example, a line can be substitute with a transparent tube. When a class sends data to be stored into the database, meshes can travel in the pipe, starting from the class reaching for the correct table. On the contrary, when classes access the database to retrieve data, meshes can travel from the table to the class.

5.2.4 Music

The visualization of the city is interesting but it can be paired with sound. By carefully analyzing the story of the system, music can be created that reflects what is happening. The different types of modifiers of the different entities can be mapped to sound and instruments. This could be combined with the live visualization, to have like a traffic noise. This can aid the understanding of the repository on different levels while entertaining the user with a nice sound.

5.3 Reflections

5.3.1 Analysis

Our entities, both software and data, are created through static analysis. This means that we do not know if the code is executed. A repository can have many classes, but use only a few. This type of information cannot be retrieved with static analysis. Differently, Dynamic Analysis analyzes the execution of the application. It can reveal which parts of the systems is primally used, which parts are rarely used and the code that is actually never used. This overcomes the evident limit of static analysis, but requires distinct approaches.

5.3.2 Model

The peculiarity of this thesis is the presence of both software and data. Currently, there are various models to represent software systems but they do not consider the data in the repository. This caused models to have classes, packages, sometimes repositories but nothing more. We introduced two different entity types versioned with the repository. By analyzing source code, we were able to construct entities that are part of the application but not versioned with the source code. Our model considers these entities and tries to make them as similar as possible to the classes and packages. The relation of a class accessing a table connects the two different parties. We decided to treat it as a single occurrence, not having versions and histories as the other entities.

5.3.3 Database

M3TRICITY 2.0 featured an inferred database, constructed with static analysis of the application source code. The static analysis has obvious limitations that restrict our knowledge of the database. However, the information we construct can be different than reality. We mark the creation of a table with the appearance of an access in the source code. We do not have knowledge about the real database. In fact it can have more tables that are not used in the analyzed repository. A table access, in reality, is a candidate table access. We do not know if the database has that table as it could be added to the application but never used.

5.3.4 Software Metrics

Software metrics are extracted equally from each class. If the system is not well constructed, this can create inconsistency in the visualization. For example, *Number of Methods* can be deceiving. A class can have few methods composed of many lines, or many methods composed of few lines. Examples are getters and setters. These type of methods are common in parts of the systems, but our analysis considers them the same as a methods of 150 lines. The logic contained is not reflected in metrics.

5.3.5 Data Metrics

We focused part of the thesis on analyzing different types of data files. We tried to represent them by extracting metrics. Our idea is that two files, written in different notations, should have the same metrics. This was not trivial. JSON and XML have different structures and different properties. The first has the structure of an object. It is created to save JavaScript instances. The opening and closing curly braces define the begging and the end of an object. The content describes the type of JavaScript class that it belongs to. The second type is more of a list of entities. It is created to store data. The opening and closing of a tag define its beginning and end and can have attributes to describe it. The content is sub-entities. JSON can also contain nested objects, as fields, and can be mixed with the primary types that the XML can have both as attributes and as sub-entities. Defining a common structure was hard but reflecting the idea in the parsers was not trivial and required extra consideration and work.

Regarding the table, normally this entity is visualized with the size of the rows as height. We did not have this information. Our knowledge was limited to the occurrence of statements in the source code trying to access them. We decided to have columns and access as metrics.

5.3.6 Visualization

Visualizing a repository and its database required many considerations. The second entity is not present in the versioning system, but it can be inferred. This makes it real and unreal at the same time. For this reason, we decided to have it see-through, hinting to the user that it is a ghost. On the other hand, its different nature should be reflected in the visualization. We initially placed them next to the city but it felt wrong. We decided to have it on a different level, allowing the user to visualize it either above or below, depending on its preference.

Bibliography

[1] A. Aiken, J. Chen, M. Stonebraker, and A. Woodruff. Tioga-2: A direct manipulation database visualization environment. In S. Y. W. Su, editor, *Proceedings of the Twelfth International Conference on Data Engineering, February 26 - March 1, 1996, New Orleans, Louisiana, USA*, pages 208–217. IEEE Computer Society, 1996.

[2] C. W. Bachman. Data structure diagrams. *Data Base*, 1(2):4–10, 1969.

[3] R. Baecker. Sorting out sorting: A case study of software visualization for teaching computer science. pages 369–381, 1998.

[4] D. Bryce and R. Hull. SNAP: A graphics-based schema manager. In *Proceedings of the Second International Conference on Data Engineering, February 5-7, 1986, Los Angeles, California, USA*, pages 151–164. IEEE Computer Society, 1986.

[5] R. G. G. Cattell. An entity-based database user interface. In P. P. Chen and R. C. Sprowls, editors, *Proceedings of the 1980 ACM SIGMOD International Conference on Management of Data, Santa Monica, California, USA, May 14-16, 1980*, pages 144–150. ACM Press, 1980.

[6] K. Chen, A. Kannan, J. Madhavan, and A. Y. Halevy. Exploring schema repositories with schemr. *SIGMOD Rec.*, 40(1):11–16, 2011.

[7] K. Chen, J. Madhavan, and A. Y. Halevy. Exploring schema repositories with schemr. In U. Çetintemel, S. B. Zdonik, D. Kossmann, and N. Tatbul, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009*, pages 1095–1098. ACM, 2009.

[8] P. P. Chen. The entity-relationship model: Toward a unified view of data. In D. S. Kerr, editor, *Proceedings of the International Conference on Very Large Data Bases, September 22-24, 1975, Framingham, Massachusetts, USA*, page 173. ACM, 1975.

[9] P. P. Chen. The entity-relationship model - toward a unified view of data. *ACM Trans. Database Syst.*, 1(1):9–36, 1976.

[10] A. Cleve, M. Gobert, L. Meurice, J. Maes, and J. H. Weber. Understanding database schema evolution: A case study. *Sci. Comput. Program.*, 97:113–121, 2015.

[11] L. M. Cortés-peña, Y. Han, N. Pradhan, and R. Rigaux. Nakedb: Database schema visualization, 2008.

[12] J. W. Davison and S. B. Zdonik. A visual interface for a database with version management. *ACM Trans. Inf. Syst.*, 4(3):226–256, 1986.

[13] S. G. Eick, J. L. Steffen, and E. E. S. Jr. Seesoft-a tool for visualizing line oriented software statistics. *IEEE Trans. Software Eng.*, 18(11):957–968, 1992.

[14] U. Erra and G. Scanniello. Towards the visualization of software systems as 3d forests: the codetrees environment. In S. Ossowski and P. Lecca, editors, *Proceedings of the ACM Symposium on Applied Computing, SAC 2012, Riva, Trento, Italy, March 26-30, 2012*, pages 981–988. ACM, 2012.

[15] F. Fittkau, A. Krause, and W. Hasselbring. Exploring software cities in virtual reality. In *3rd IEEE Working Conference on Software Visualization, VISSOFT 2015, Bremen, Germany, September 27-28, 2015*, pages 130–134. IEEE Computer Society, 2015.

[16] F. Fittkau, J. Waller, C. Wulf, and W. Hasselbring. Live trace visualization for comprehending large software landscapes: The explorviz approach. In A. Telea, A. Kerren, and A. Marcus, editors, *2013 First IEEE Working Conference on Software Visualization (VISSOFT), Eindhoven, The Netherlands, September 27-28, 2013*, pages 1–4. IEEE Computer Society, 2013.

[17] H. C. Gall, M. Jazayeri, and C. Riva. Visualizing software release histories: The use of color and third dimension. In *1999 International Conference on Software Maintenance, ICSM 1999, Oxford, England, UK, August 30 - September 3, 1999*, pages 99–108. IEEE Computer Society, 1999.

[18] T. A. Girba. *Modeling history to understand software evolution*. PhD thesis, University of Bern, 2005.

[19] O. Greevy, M. Lanza, and C. Wysseier. Visualizing live software systems in 3d. In E. T. Kraemer, M. M. Burnett, and S. Diehl, editors, *Proceedings of the ACM 2006 Symposium on Software Visualization, Brighton, UK, September 4-5, 2006*, pages 47–56. ACM, 2006.

[20] L. M. Haibt. A program to draw multilevel flow charts. In *Papers Presented at the the March 3-5, 1959, Western Joint Computer Conference*, IRE-AIEE-ACM '59 (Western), page 131–137, New York, NY, USA, 1959. Association for Computing Machinery.

[21] R. C. Holt and J. Y. Pak. GASE: visualizing software evolution-in-the-large. In *3rd Working Conference on Reverse Engineering, WCRE '96, Monterey, CA, USA, November 8-10, 1996*, page 163. IEEE Computer Society, 1996.

[22] D. A. Keim. Pixel-oriented database visualizations. *SIGMOD Rec.*, 25(4):35–39, 1996.

[23] D. A. Keim and H. Kriegel. Visdb: database exploration using multidimensional visualization. *IEEE Computer Graphics and Applications*, 14(5):40–49, 1994.

[24] D. A. Keim and H. Kriegel. Visdb: A system for visualizing large databases. In M. J. Carey and D. A. Schneider, editors, *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, USA, May 22-25, 1995*, page 482. ACM Press, 1995.

[25] C. Knight and M. Munro. Virtual but visible software. In *International Conference on Information Visualisation, IV 2000, London, England, UK, July 19-21, 2000*, pages 198–205. IEEE Computer Society, 2000.

[26] D. E. Knuth. Computer-drawn flowcharts. *Commun. ACM*, 6(9):555–563, 1963.

[27] G. Langelier, H. A. Sahraoui, and P. Poulin. Visualization-based analysis of quality for large-scale software systems. In D. F. Redmiles, T. Ellman, and A. Zisman, editors, *20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005), November 7-11, 2005, Long Beach, CA, USA*, pages 214–223. ACM, 2005.

[28] M. Lanza. Combining metrics and graphs for object oriented reverse engineering, 1999.

[29] M. Lanza. The evolution matrix: Recovering software evolution using software visualization techniques. In T. Tamai, editor, *Proceedings of the 4th International Workshop on Principles of Software Evolution, IWPSE 2001, Vienna, Austria, September 10-11, 2001*, pages 37–42. ACM, 2001.

[30] C. Marinescu. Applications of automated model's extraction in enterprise systems. In M. van Sinderen and L. A. Maciaszek, editors, *Proceedings of the 14th International Conference on Software Technologies, ICSOFT 2019, Prague, Czech Republic, July 26-28, 2019*, pages 254–261. SciTePress, 2019.

[31] K. Maruyama, T. Omori, and S. Hayashi. A visualization tool recording historical data of program comprehension tasks. In C. K. Roy, A. Begel, and L. Moonen, editors, *22nd International Conference on Program Comprehension, ICPC 2014, Hyderabad, India, June 2-3, 2014*, pages 207–211. ACM, 2014.

[32] C. Mesnage and M. Lanza. White coats: Web-visualization of evolving software in 3d. In S. Ducasse, M. Lanza, A. Marcus, J. I. Maletic, and M. D. Storey, editors, *Proceedings of the 3rd International Workshop on Visualizing Software for Understanding and Analysis, VISSOFT 2005, Budapest, Hungary, September 25, 2005*, pages 40–45. IEEE Computer Society, 2005.

[33] L. Meurice and A. Cleve. DAHLIA: A visual analyzer of database schema evolution. In S. Demeyer, D. W. Binkley, and F. Ricca, editors, *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering, CSMR-WCRE 2014, Antwerp, Belgium, February 3-6, 2014*, pages 464–468. IEEE Computer Society, 2014.

[34] L. Meurice and A. Cleve. DAHLIA 2.0: A visual analyzer of database usage in dynamic and heterogeneous systems. In B. Sharif, C. Parnin, and J. Fabry, editors, *2016 IEEE Working Conference on Software Visualization, VISSOFT 2016, Raleigh, NC, USA, October 3-4, 2016*, pages 76–80. IEEE Computer Society, 2016.

[35] H. A. Müller and K. Klashinsky. Rigi - A system for programming-in-the-large. In T. C. Nam, L. E. Druffel, and B. Meyer, editors, *Proceedings, 10th International Conference on Software Engineering, Singapore, Singapore, April 11-15, 1988*, pages 80–87. IEEE Computer Society, 1988.

[36] C. Nagy and A. Cleve. Sqlinspect: a static analyzer to inspect database usage in java applications. In M. Chaudron, I. Crnkovic, M. Chechik, and M. Harman, editors, *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 93–96. ACM, 2018.

[37] I. Nassi and B. Shneiderman. Flowchart techniques for structured programming. *SIGPLAN Not.*, 8(8):12–26, Aug. 1973.

[38] T. Panas, R. Berrigan, and J. C. Grundy. A 3d metaphor for software production visualization. In E. Banissi, K. Börner, C. Chen, G. Clapworthy, C. Maple, A. Lobben, C. J. Moore, J. C. Roberts, A. Ursyn, and J. J. Zhang, editors, *Seventh International Conference on Information Visualization, IV 2003, 16-18 July 2003, London, UK*, pages 314–319. IEEE Computer Society, 2003.

[39] T. Panas, T. Epperly, D. J. Quinlan, A. Sæbjørnsen, and R. W. Vuduc. Communicating software architecture using a unified single-view visualization. In *12th International Conference on Engineering of Complex Computer Systems (ICECCS 2007), 10-14 July 2007, Auckland, New Zealand*, pages 217–228. IEEE Computer Society, 2007.

[40] T. Panas, R. Lincke, and W. Löwe. Online-configuration of software visualizations with vizz3d. In T. L. Naps and W. D. Pauw, editors, *Proceedings of the ACM 2005 Symposium on Software Visualization, St. Louis, Missouri, USA, May 14-15, 2005*, pages 173–182. ACM, 2005.

[41] F. Pfahler, R. Minelli, C. Nagy, and M. Lanza. Visualizing evolving software cities. In *Working Conference on Software Visualization, VISSOFT 2020, Adelaide, Australia, September 28 - October 2, 2020*, pages 22–26. IEEE, 2020.

[42] F. Pfhaller. M3tricity: A 3d evolution-resistant visualization of software systems. Master's thesis, USI Università della Svizzera italiana, 2020.

[43] S. P. Reiss. An engine for the 3d visualization of program information. *J. Vis. Lang. Comput.*, 6(3):299–323, 1995.

[44] T. R. Rogers and R. G. G. Cattell. Entity-relationship database user interfaces. In S. T. March, editor, *Entity-Relationship Approach, Proceedings of the Sixth International Conference on Entity-Relationship Approach, New York, USA, November 9-11, 1987*, pages 353–365. North-Holland, 1987.

[45] T. R. Rogers and R. G. G. Cattell. Entity-relationship database user interface. *IEEE Data Eng. Bull.*, 11(2):44–53, 1988.

[46] A. Schreiber and M. Bruggemann. Interactive visualization of software components with virtual reality headsets. In *IEEE Working Conference on Software Visualization, VISSOFT 2017, Shanghai, China, September 18-19, 2017*, pages 119–123. IEEE, 2017.

[47] F. Steinbrückner and C. Lewerentz. Representing development history in software cities. In A. Telea, C. Görg, and S. P. Reiss, editors, *Proceedings of the ACM 2010 Symposium on Software Visualization, Salt Lake City, UT, USA, October 25-26, 2010*, pages 193–202. ACM, 2010.

[48] M. Stonebraker, J. Chen, N. Nathan, C. Paxson, A. Su, and J. Wu. Tioga: A database-oriented visualization tool. In G. M. Nielson and R. D. Bergeron, editors, *4th IEEE Visualization Conference, IEEE Vis 1993, San Jose, CA, USA, October 25-29, 1993, Proceedings*, pages 86–93. IEEE Computer Society, 1993.

[49] M. Stonebraker, J. Chen, N. Nathan, C. Paxson, and J. Wu. Tioga: Providing data management support for scientific visualization applications. In R. Agrawal, S. Baker, and D. A. Bell, editors, *19th International Conference on Very Large Data Bases, August 24-27, 1993, Dublin, Ireland, Proceedings*, pages 25–38. Morgan Kaufmann, 1993.

[50] M. Stonebraker and L. A. Rowe. Database portals: A new application program interface. In U. Dayal, G. Schlageter, and L. H. Seng, editors, *Tenth International Conference on Very Large Data Bases, August 27-31, 1984, Singapore, Proceedings*, pages 3–13. Morgan Kaufmann, 1984.

[51] Y. Tymchuk, A. Mocci, and M. Lanza. Vidi: The visual design inspector. In A. Bertolino, G. Canfora, and S. G. Elbaum, editors, *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 2*, pages 653–656. IEEE Computer Society, 2015.

[52] R. Wettel. *Software Systems as Cities*. PhD thesis, Università della Svizzera Italiana, 2010.

[53] R. Wettel and M. Lanza. Visualizing software systems as cities. In J. I. Maletic, A. Telea, and A. Marcus, editors, *Proceedings of the 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis, VISSOFT 2007, Banff, Alberta, Canada, June 25-26, 2007*, pages 92–99. IEEE Computer Society, 2007.

[54] H. K. T. Wong and I. Kuo. GUIDE: graphical user interface for database exploration. In *Eigth International Conference on Very Large Data Bases, September 8-10, 1982, Mexico City, Mexico, Proceedings*, pages 22–32. Morgan Kaufmann, 1982.

[55] C. Zirkelbach and W. Hasselbring. Live visualization of database behavior for large software landscapes: The raccoon approach. 2019.