CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Exploring Software Evolution with Class Role Stereotype Visualisation

Master's thesis in Software Engineering

KIN YAN LEE

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2021

Master's thesis 2021

Exploring Software Evolution with Class Role Stereotype Visualisation

KIN YAN LEE

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2021

Exploring Software Evolution with Class Role Stereotype Visualisation

KIN YAN LEE

Supervisor: Truong Ho-Quang, Department of Computer Science and Engineering
Examiner: Jennifer Horkoff, Department of Computer Science and Engineering

Master's Thesis 2021
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Gothenburg, Sweden 2021

Exploring Software Evolution with Class Role Stereotype Visualisation
KIN YAN LEE
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

A major challenge for software developers in an organisation is to understand the software systems so as to perform software evolution tasks. The conventional approach is to examine the code, which is often arduous and time-consuming. On the other hand, visualisation is often viewed as a better way to represent data. As the usefulness of class role stereotypes has been investigated, this study is conducted to shed light on the use of evolution data of role stereotypes in visualisation in relation to performing software evolution tasks. This study aims to develop a visualisation approach, which focuses on the evolution of class role stereotypes in object-oriented software systems, using the design science research approach, along with a user study evaluating the approach in performing software evolution tasks. The user study was conducted with six participants with the context of utilising the tool in the understanding part of evolution tasks, and they were asked to work on two tasks, one with the visualisation and other without. Qualitative analysis was carried out on the data collected from the survey and the video recordings of the tasks. The analysis shows the use of the visualisation tool, named Rologram, appears to help identify the changes of responsibility and collaborations of the classes, in comparison to the approach without. It is concluded that the visualisation approach tends to be helpful in performing software evolution tasks.

Keywords: software evolution, data visualisation, object-oriented programming, class role stereotypes, software maintenance.

Acknowledgements

I would like to express my sincere gratitude to my supervisor Truong Ho-Quang for his guidance and support in particular. I am also grateful to the experts who provided constructive feedback on the visualisation, as well as the participants in the user study.

Kin Yan Lee, Gothenburg, June 2021

Contents

List of Figures

List of Tables

1

Introduction

1.1 Context

Software evolution involves two major areas: design for changing a software system over time and analysis of software histories to detect regularities and anomalies (Diehl, 2007, p. 129). Analysing the software evolution data in order to better perform changes to software systems is not uncommon, especially structural changes. Developers that are new to the systems likely find it arduous to work on software evolution tasks. In this sense, software evolution visualisation helps present the massive amount of data in a more accessible and understandable way, which facilitates the developers in performing those tasks.

Visualising software evolution data is a subject that is studied by many researchers with proposed visualisation tools of software evolution (Eick et al., 2002; Holt & Pak, 1996; Lanza & Ducasse, 2002; Nam et al., 2018; Rysselberghe & Demeyer, 2004). There are also numerous studies focusing on software visualisation techniques (Bani-Salameh et al., 2016; Kienle & Müller, 2007) and interactions in information visualisation (Yi et al., 2007).

A software evolution task often involves comprehension of the software system. The six role stereotypes proposed by Wirfs-Brock (2006) help understand the responsibility of the classes in object-oriented software systems. Role stereotypes with dependencies between the classes show the relationships and interactions between different roles, for example, a class with the Information Holder role provides information to a Controller class. By getting the information of role stereotypes of all classes and how they collaborate, developers may tell the structure of a system. During the software evolution process, refactoring and alterations to the system structure are often inevitable, for instance, splitting a large class into several smaller classes, role stereotype changes are probably involved as the class structures are modified and dependencies to other classes are created. Change of role stereotypes can be seen in 5 to 14% of the classes in each of the three software systems investigated in the study of the evolution of role stereotypes (Fröding & Ngoc, 2020).

Design degradation is not uncommon as a side effect of software evolution since developers may not have the original designs in mind when fixing bugs and enhancing parts of the system. With the evolution data of role stereotypes, developers can possibly spot changes to system structures, which may help identify any incon-

sistencies and contradictions to the designs or evaluate whether a redesign is needed.

The study of Ho-Quang et al. (2020) demonstrates the usefulness of role stereotypes in comprehending software systems with the proposed visualisation tool RoleViz. Fröding and Ngoc (2020) study the evolution of role stereotypes and the anti-patterns identified in relation to them in three open-source software systems. With these studies as inspirations, it is interesting to look into whether visualising changes of role stereotypes could help developers perform software evolution tasks.

1.2 Statement of the Problem

Visualising software data such as the numbers of methods, attributes and lines of code has been explored by numerous studies. In comparison with visualising these measurements, studies on role stereotypes, which are important information with a higher abstraction level that potentially help developers understand the system structures and class behaviours, are scarce. There are currently no software evolution visualisation tools showing the class role stereotypes available, and little research investigating the usefulness of the changes of class role stereotypes.

As a software system evolves, huge costs and efforts are needed to understand how the system changes. This raises the questions of whether an interactive visualisation tool highlighting the changes of class role stereotypes in the software evolution process could help reduce the time and efforts when working on software evolution tasks and how.

1.3 Purpose of the Study

The purpose of this study is to validate the usefulness of software evolution visualisation focusing on class role stereotypes by creating an interactive visualisation and conducting a user study to evaluate how the visualisation can help developers understand the software systems and perform evolution tasks.

1.4 Contribution of the Study

The study intends to propose and evaluate an interactive visualisation approach for developers performing software evolution tasks, showing software evolution data with focus on class role stereotypes. The main contributions of this study are the visualisation approach demonstrating the evolution of software systems, particularly the changes of class role stereotypes, the evaluation of the tool with the scenario of developers performing software evolution tasks and the approaches used with and

without the visualisation tool.

IT practitioners and software developers can benefit from the results of this study if the visualisation of evolution of class role-stereotypes helps perform software evolution tasks and understand software systems.

The study can be beneficial for researchers considering it would be an empirical piece added in studying the usefulness of visualising class role stereotypes in relation to software evolution tasks. This study would also provide researchers with data and knowledge on the visualisation approach and the evaluation.

1.5 Structure of the Paper

This report consists of nine chapters. Chapter 1 introduces the problem to be addressed, purpose and the contribution of the study. Chapter 2 discusses the background and related work on role stereotypes and their usefulness, visualisation techniques, as well as existing role stereotype visualisation and software evolution visualisations. The research questions are presented in Chapter 3. Chapter 4 explains the methods adopted in this study. Chapter 5 introduces the visualisation approach and the implementation. Chapter 6 describes the results of the study. Chapter 7 presents the answers to the research questions. Chapter 8 discusses the limitations and delimitations of the study. Lastly, Chapter 9 concludes the study and mentions the future work.

2

Background and Related Work

This chapter provides some background knowledge on role stereotypes and introduces the related work with focus on the usefulness of role stereotypes, techniques for information visualisation, and existing visualisation tools on role stereotypes and software evolution.

2.1 Role Stereotypes

Responsibility-driven design proposed by Wirfs-Brock (1990) is an approach of object-oriented design. She defines role as a set of relevant responsibilities, responsibility as an obligation to perform a task or know information, and collaboration as an interaction of objects or/and roles (Wirfs-Brock & McKean, 2002). An object takes a specific role at a time and fulfils a responsibility with or without collaborating with other objects; the collaboration patterns disclose the control and information flow in the system, such that misassignment of responsibility to a class can be revealed (Wirfs-Brock, 1990, pp. 90-91).

The notion of role stereotypes is fundamental to this study. Wirfs-Brock (2006) identifies six role stereotypes as simplified roles that carry predefined responsibilities for classes and interfaces in object-oriented software systems, namely Controller, Coordinator, Information Holder, Interfacer, Service Provider and Structurer. Not only do these role stereotypes help understand the class behaviours, but also the design patterns adopted in the software systems (Wirfs-Brock, 2006). Table 2.1 lists the role stereotypes with description of their responsibilities.

Role stereotype	Responsibilities
Controller	To make decisions and control complicated tasks.
Coordinator	To delegate work to other objects rather than making lots of decisions.
Information Holder	To know information and supply it to other objects.
Interfacer	To transform requests or information between different parts of a system. To interact with the user or communicate with external systems or between subsystems.
Service Provider	To perform certain work and provide services to other objects on demand.
Structurer	To maintain relationships between various objects and hold information regarding these relationships.

Table 2.1: Six role stereotypes characterising the responsibility of an object

2.2 Usefulness of Role Stereotypes

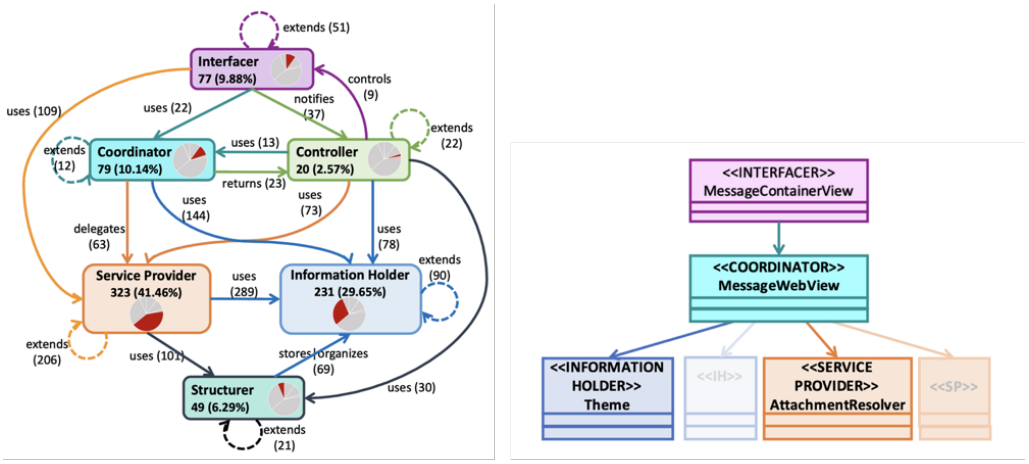
Ho-Quang et al. (2020) create the tool RoleViz visualising role stereotypes and compare it to the existing industrial visualisation tool Softagram, and it appears that the participants using RoleViz had better performance than those with Softagram when carrying out software comprehension tasks in the user study.

Fröding and Ngoc (2020) investigate the changes of role stereotypes among different versions of three software systems and adopt the data to identify anti-patterns. The distribution of role stereotypes has changed remarkably between several versions in one of the software systems as shown in the results of their study. It is found that the distribution of anti-patterns in certain role stereotypes seems to have connection with the changes of the roles' distribution.

Their study on the evolution of role stereotypes (Fröding & Ngoc, 2020), is based on the data of three open-source software systems generated from a role stereotype classifying tool using machine learning, namely Class Role Identifier (Ho-Quang et al., n.d., as cited in Fröding & Ngoc, 2020). The tool provides 74% and 88% accuracy rate using the Random Forest algorithm without and with the use of the synthetic minority oversampling technique (SMOTE) respectively for the classification of different role stereotypes (Ho-Quang et al., n.d.).

Nurwidyantoro et al. (2019) illustrate applications of role stereotypes, one of which is the patterns of collaboration between role stereotypes. Figure 2.1 (a) shows the relationships between the role stereotypes in the K-9 Mail software system, in which the most common collaboration is Service Provider with Information Holder. Figure 2.1 (b) shows a common collaboration pattern between role stereotypes in the system, where Interfacer reacts to events of user interface by communicating to Coordinator,

and then the Coordinator divides the tasks and forwards the information/requests to the associated Service Provider and Information Holder classes (Nurwidyantoro et al., 2019). Recurring collaboration patterns indicate the regularities in the design of software systems (Nurwidyantoro et al., 2019).



(a) (b)

Figure 2.1: Relationships between role stereotypes (a) and a common collaboration pattern between role stereotypes (b) in K-9 Mail

2.3 Interaction and Software Visualisation Techniques

Yi et al. (2007) emphasise on the importance of interaction components in information visualisation and propose seven categories of interaction techniques, i.e. select, explore, reconfigure, encode, abstract/elaborate, filter and connect, in supporting user intents. Table 2.2 includes the description for the categories.

Category	Description
Select	Mark a specific item
Explore	Show subset of data
Reconfigure	Arrange in different way
Encode	Display in different representations
Abstract/Elaborate	Adjust data abstraction levels
Filter	Change according to different conditions
Connect	Display only relevant items

Table 2.2: Seven categories of interaction techniques

Bani-Salameh et al. (2016) propose six functional requirements for software evolution visualisation tools, including views, details-on demand, filter and search, select, re-arrangement, and comparison, giving the ideas of the necessary functionalities a

software evolution visualisation should have. Kienle and Müller (2007) identify the key quality attributes and functional requirements of software visualisation tools, which comprise views, abstraction, search, filters, code proximity, automatic layouts, undo/history and miscellaneous.

2.4 Existing Role Stereotype Visualisation Tool – RoleViz

RoleViz is the only visualisation available visualising the role stereotypes of the classes in software systems (Figure 2.2). Polymetric view is used in the visualisation, with the metrics of fan-in, i.e. the number of units depends on the unit (height), fan-out, i.e. the number of units the unit depends on (width) and role stereotype (colour) incorporated in the compilation units, which are represented by coloured box inside a larger grey box (package). Package-level dependencies are illustrated by bimetric lines showing the number of dependencies between the packages in both directions. The visualisation has also adopted some interaction techniques including hovering over a package to view its full path and get the relevant dependencies highlighted.

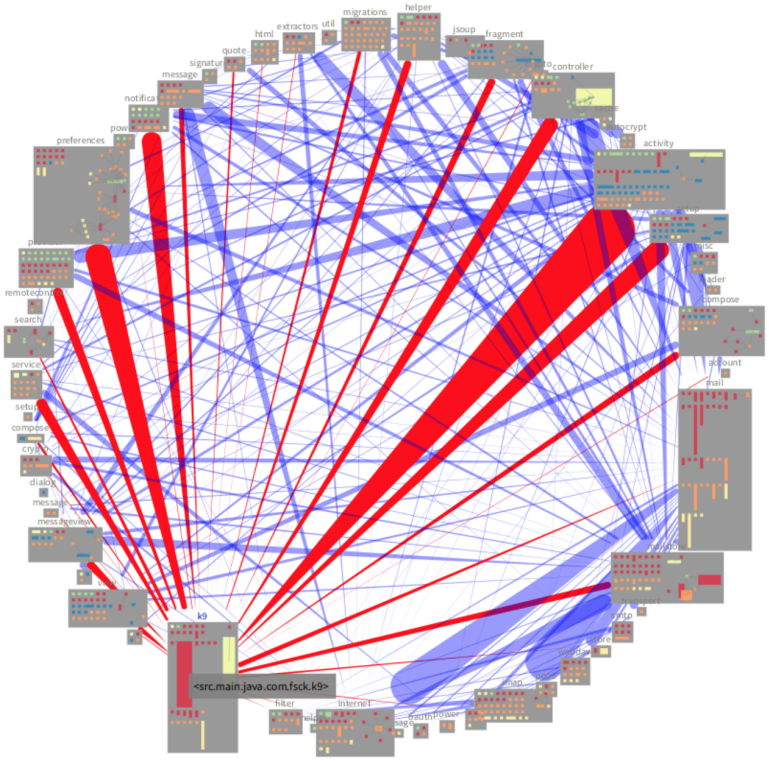


Figure 2.2: RoleViz visualising K-9 Mail with role stereotypes

The visualisation proposed in this paper basically adopts the same colours used in RoleViz to encode class role stereotypes, with some adjustments made. The

intention of RoleViz is to visualise only one specific version of an object-oriented software system, incapable of illustrating the evolution of a system. On the contrary, the focus of the proposed visualisation approach is on the evolution of a software system.

2.5 Alternatives of Software Evolution Visualisation Tools

2.5.1 Visualising Change History

Rysselberghe and Demeyer (2004) introduce a visualisation on the evolution of the open-source system Tomcat based on the information extracted from Concurrent Versions System (CVS) including revision number, file name, author, date, time, lines changed and log message for each revision. Time and files are the two dimensions used in the visualisation. The files are ordered by the full path name. A dot implies a change committed for one file at a specific time. The vertical line pattern indicates that the file has changed frequently across the revisions. Figure 2.3 shows the evolution of the change history of all files in Tomcat from 1999 to 2004.

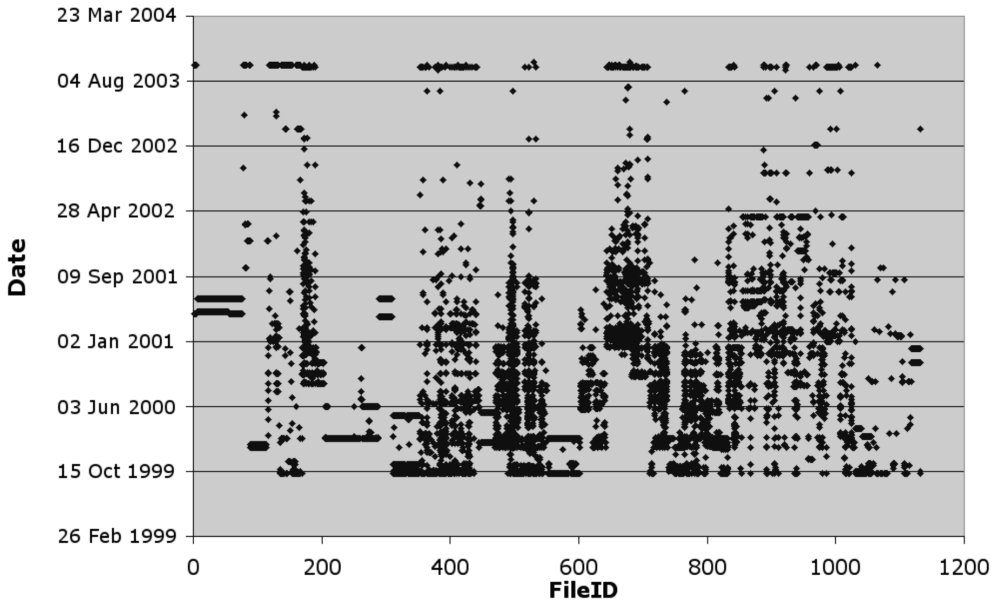


Figure 2.3: Visualisation of evolution of Tomcat using change history

It is interpreted that badly designed components can be located by more dense lines in the visualisation as frequent changes are needed in order to synchronise with other parts of the system, and coherent entities can be identified by similar change patterns as the entities are changed simultaneously (Rysselberghe & Demeyer, 2004). Moreover, horizontal lines with no or little dots indicate a low productivity of the development team.

The visualisation proposed by Rysselberghe and Demeyer (2004) is capable of visualising the evolution of the software system using a single diagram; however, it only focuses on the change history and lacks the evolution of the system structure.

2.5.2 Visualising Software Architecture - EVA

EVA is a visualisation tool that illustrates software evolution with the aspect of software architecture and supports three different kinds of views: Single-Release Architecture (see Figure 2.4 (a)), 3-D Architecture-Evolution (see Figure 2.4 (b)) and Pairwise Architecture-Comparison (see Figure 2.4 (c)) (Nam et al., 2018). The tool extracts the issue data from the user-specified repository and uses them to map code-level entities to the corresponding architectural components. As shown in Figure 2.4, smaller coloured circles represent code-level entities, whereas the larger circles depict architectural components. The code-level entities from the same package are encoded with the same colour. Stacked layers and lines are used to illustrate the changes from multiple versions, and the pairwise changes including the addition and deletion of code-level entities are annotated with labels on the circles.

One can locate the changes of entity existence by the labels inside the circles which is efficient. The visualisation provides comparison of multiple versions, nevertheless, one needs to hover over each circle to view the changes and get the information of the entities, in which a considerable number of user actions is required in order to locate an entity or view the changes of all the entities.

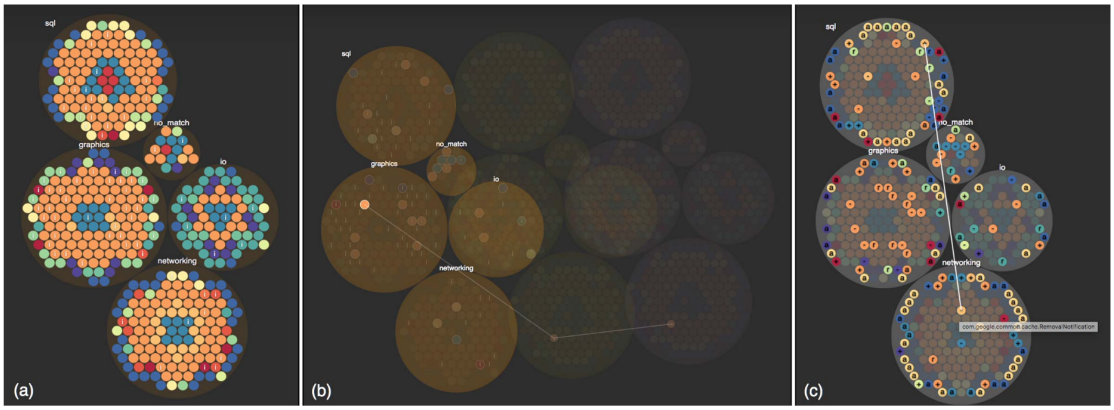


Figure 2.4: Three types of views in EVA

3
Research Questions

The study involves two research questions regarding software evolution visualisation (RQ1) and evaluation of the visualisation approach (RQ2).

RQ 1: *How can data of evolution of class role stereotypes of a software system be visualised?*
This aims to suggest an approach for visualising the changes of class role stereotypes. The input of class role stereotype data is from the related studies, whereas other relevant software evolution data is extracted with tools. As an outcome of answering RQ1, an interactive visualisation tool that visualises the evolution data of role stereotypes change is built.

RQ 2: *How does the proposed visualisation approach/tool help developers perform software evolution tasks?*
This part aims to investigate and evaluate the visualisation approach from RQ1 with the following sub questions.
SQ 2.1: *Is the proposed visualisation approach/tool helpful?*
SQ 2.2: *What approach do developers use? (with and without the tool)*
SQ 2.3: *How can we improve Rologram?*

4
Methods

4.1 Research Process

This study adopts the design science research approach for the implementation of the visualisation (to answer RQ1) and a user study was conducted as part of the design science research and also a qualitative research (to answer RQ2). Figure 4.1 illustrates the process of this study.

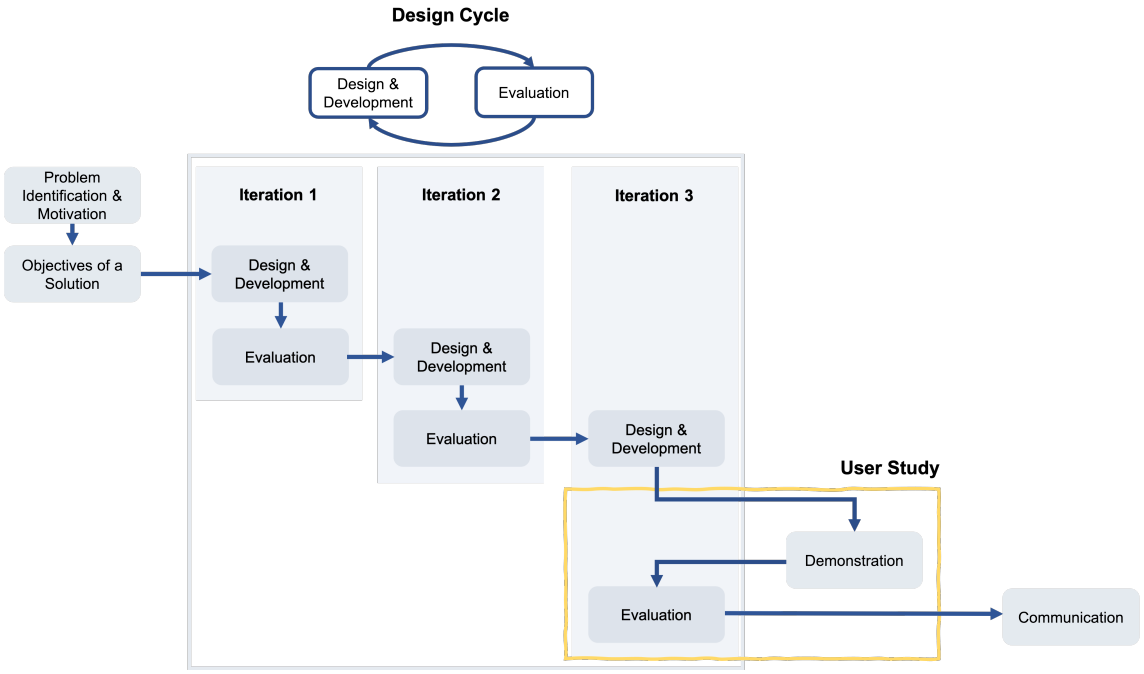


Figure 4.1: Research process

Section 4.2 describes the design science methods used to build the visualisation tool (Rologram). Section 4.3 describes the recruitment of participants, tasks, procedures, data collection and analysis of the user study.

4.2 Design Science Research Approach

The methodology for design science research proposed by Peffers et al. (2006, 2007) was adopted and the process model consisting of problem identification and motivation, objectives of a solution, design and development, evaluation, demonstration,

and communication was followed. The activities in the process are described chronologically as follows.

Problem identification and motivation. The problem-centred approach was adopted as the research was initiated by the problem identified in Chapter 1 (Peffers et al., 2007). The motivation and contribution are also mentioned in Chapter 1. Moreover, preparations for the data inputs were carried out prior to the step of design and development. The datasets of class role stereotypes from the study of Fröding and Ngoc (2020) were reused as the data inputs for the artefact, including K-9 Mail, with 37 commits ranging from February 2011 to February 2020, and Bitcoin Wallet, with 35 commits ranging from March 2011 to January 2020. The static dependencies between classes were extracted from the source code downloaded from GitHub using the Java class dependency analyser (jdeps).

Objectives of a solution. The objective was to develop a solution of visualising the evolution of class role stereotypes of object-oriented software systems that intends to help developers to comprehend the systems and hence accomplish software evolution tasks.

According to Hevner (2007), a design cycle iterates the research activities of building and evaluation of the design artefacts and processes with refinement based on feedback. A design science research must produce a workable artefact (Hevner et al., 2004). The artefact, i.e. the visualisation tool, has gone through three iterations of design cycle.

1. First iteration:
   Design and development. The functionalities of the artefact for the solution were analysed at the beginning and implemented.
   Evaluation. The aim of the evaluation was to improve the artefact. Reviews from three experts in software architecture, evolution and class role stereotypes, which were also the authors of a study on role stereotypes, software architecture and visualisation (Ho-Quang et al., 2020), were conducted by sending the link to the artefact through emails. The feedback on the artefact from the experts was collected and examined to see if it was feasible to implement in the next iteration. A list of the feedback and the corresponding implementation plan was sent to the reviewers for informing and verification purposes.

2. Second iteration:
   Design and development. The artefact was refined, and more functionalities were added and enhanced with considerations of the feedback from the evaluation in the first iteration.
   Evaluation. The aim and procedure for this evaluation were the same as the first one. Reviews from three experts in software architecture, evolution and class role stereotypes were conducted through email communications, two of which participated in the first evaluation. Their feedback on the artefact was

collected and examined. One of the experts also provided extra feedback on the artefact during an online meeting.

3. Third iteration: A user study served as both demonstration and final evaluation was conducted in this iteration, with the purpose of evaluating how the actual use of artefact in the demonstration supports the objectives of the solution. The details of the user study are described in Section 4.3.
   Design and development. The feedback collected from the previous evaluations was taken into account during the development of the artefact.
   Demonstration. The resulting artefact was used by the participants in the user study, by working on the specific tasks with the context of understanding the evolution of the Bitcoin Wallet system.
   Evaluation. The user study was conducted in order to investigate how the artefact could help solve the identified problem.

Communication. This report serves as the communication of the research, reporting all the activities and the results of the research process.

4.3 User Study

This section describes the user study carried out as an evaluation of the design science research.

4.3.1 Data Collection

The participants were expected to complete the given tasks and provide the answers in the questionnaire. During the task completion, the operations and behaviours of the participants were screen recorded, and the video recordings were collected as qualitative data. The recordings only contained the video part without audio.

The questionnaire, which can be found in Appendix, contained both qualitative and quantitative questions. Questions related to demographics, equipment, approaches used and feedback on visualisation were asked in the format of qualitative questions, whilst the quantitative questions were in the form of the 5-point Likert rating scale (from strongly disagree to strongly agree). Both qualitative and quantitative questions regarding the tasks, usefulness of role stereotypes and the visualisation were formulated. Multiple-choice questions were not considered for the tasks so as to prevent the participants from randomly picking an answer, which could affect the results. The time taken for completing each task was collected. The questionnaire was provided to the participants through Google Form.

Qualitative data was collected through the questionnaire with the following questions:

- Elaborate on how role stereotypes help complete the tasks.
- Elaborate on how the visualisation helps complete the tasks.
- Describe the approach with the visualisation.

- Describe the approach without the visualisation.
- Please comment on the visualisation regarding usability and other aspects you could think of.
- Please provide some suggestions to improve the visualisation. Think of any missing features you would like to have.

4.3.2 Tasks

The aim was to evaluate the tool with real evolution tasks. In reality, the real evolution tasks might comprise both understanding and development parts. In this user study, with the time restriction of 35 minutes per task, the focus was on the understanding part of the evolution task. It is considered as sufficient for this study owing to the fact that understanding is an utmost important part for developers to perform any development/evolution activities.

The data of Bitcoin Wallet and K-9 Mail was used in this study. The Bitcoin Wallet system was chosen for the tasks since it contained comparatively less amount of classes and packages compared to that of K-9 Mail system, and hence the time needed and complexity of the tasks could be reduced.

The participants were given two different evolution tasks of the Bitcoin Wallet software system. Each participant performed one of the two tasks with help of the visualisation tool (Rologram) and the other task without the visualisation tool (the participant was not restricted in the tools or techniques to be used).

Each task was to investigate the evolution of a specific class and package in the Bitcoin Wallet system across five commit versions, in the aspects of the functionalities of the class, and class- and package-level dependencies. The two tasks targeted different parts of the Bitcoin Wallet system (i.e. different packages); thus, a low learning effect was expected from performing one task to apply on the other task. Two packages that were of comparable size in terms of the numbers of classes within the package and of the package-level dependencies for the two tasks were chosen. The number of class-level dependencies for the two selected classes was also comparable; therefore, the same amount of workload was expected when performing the tasks. The questions for the tasks can be found in the questionnaire in Appendix.

The sorts of the questions in each task are as follow.

1. Find two changes of class-level dependencies between the specific package and any of other packages between two commits.
2. Identify the changes of package-level dependencies centred the specific package over the five commits.
3. Identify the functionality/responsibility of the targeted class in the specific commit.
4. Identify the major changes of the targeted class in terms of its functionality/responsibility over the five commits.

The participants were randomly assigned with different combinations of the tasks and tools. The maximum time for completing each task was restricted to 35 minutes and the time taken for each task was recorded.

After data collection from the survey, the answers for the task questions were marked and given points according the definite solution found by examination of the software system using Eclipse.

4.3.3 Participants

The sample size was 6. The non-probability sampling method convenience sampling was adopted to select the participants for user study as they were searched and recruited through the personal network for higher accessibility and response rate.

The visualisation only supports Java projects at the moment as the software systems selected in the input datasets are written in Java; accordingly, the target population of the user study was developers with Java programming experience. The participants must meet the criterion of having some Java programming experience. Actual working experience in the industry was preferred but not necessary. The exclusive criterion was the participants were not contributors of the Bitcoin Wallet system, which was used in the user study, so as to avoid the situation that they could answer the questions in the tasks directly, which might affect the results.

4.3.4 Procedure

The activities before and during a user study are illustrated in Figure 4.2. Before a user study, participants were contacted and details of the user study along with the link to the trial version of the visualisation with the data of K-9 Mail software system, which was not used in the tasks to avoid the participants got familiarised with the Bitcoin Wallet system before the user study, and introduction of role stereotypes were sent once they agreed to participate in user study. Each participant was scheduled with a time to conduct a user study.

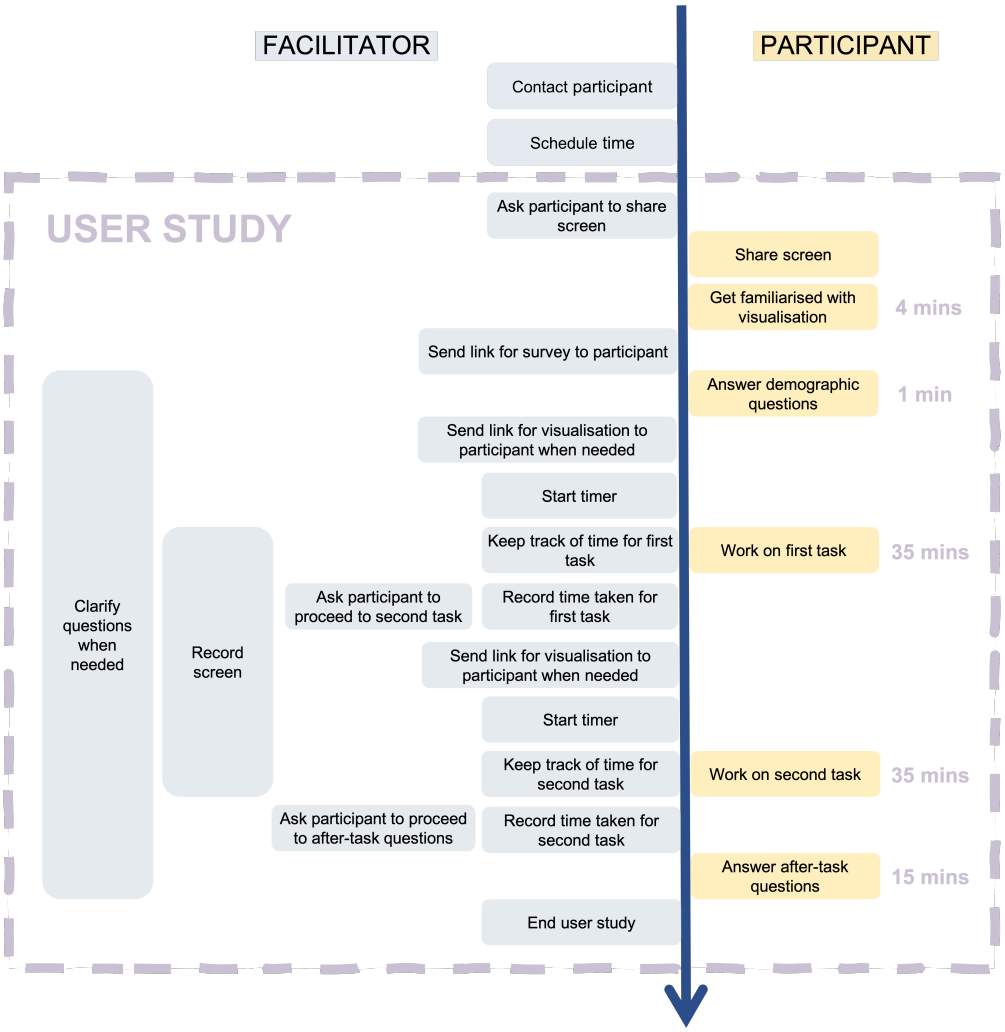


Figure 4.2: Procedure of conducting the user study with one participant

The user study was conducted remotely with video conferencing software with one participant each time, and the duration was restricted to a maximum of one and a half hours. The activities in a user study in chronological order were as follows. The participant was requested to share the screen with the facilitator and given time to get familiarised with the trial version of the visualisation (4 minutes). The facilitator then sent the link to the survey to the participant and the participant can start to provide some background information in the first section of the survey (1 minute). Communication with the participant during the whole user study was maintained for clarifying the questions. After completing the demographic questions, the participant can start working on the two tasks and the screen for the whole process of the two tasks was recorded. One participant refused to share his screen and thereby no corresponding video recording was collected. The link to the visualisation tool with data of the Bitcoin Wallet system was sent when the participant was ready to work on the task with visualisation.

The facilitator started the timer and the participant worked on the first task, with

a time restriction of 35 minutes. The participant may complete a task with less than 35 minutes. After 35 minutes, the participant was requested to proceed to the second task. The time taken for the first task was recorded. The procedure for the second task was handled in the same way. No break was given in between the two tasks unless the participant requested. The video recording of the participant working on the tasks and the two time for the duration of working on the first and second tasks were collected. The participant had to answer the after-task questions in the survey (15 minutes) after working on the second task. After the participant submitted the survey, the demographic, qualitative and quantitative data in the questionnaire was gathered.

4.3.5 Data Analysis

The steps for qualitative data analysis in the book written by Auerbach and Silverstein (2003) were followed (p. 43). The inputs and outputs produced for each step are illustrated by Figure 4.3.

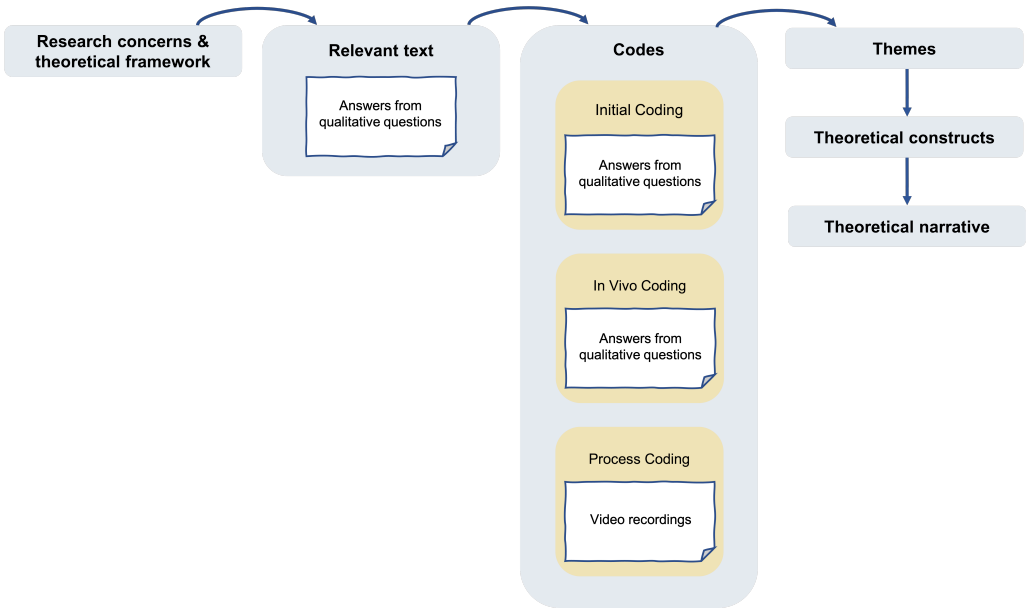


Figure 4.3: Inputs and outputs for the steps in data analysis

Firstly, the research concern was about how the visualisation tool helped developers perform software evolution tasks and the theoretical framework was discussed in Chapter 2. Secondly, the relevant text was selected from the answers collected in the survey. Thirdly, the related relevant text was grouped together to derive the repeating ideas (codes) using the coding methods Initial Coding and In Vivo Coding, and the behaviours of research participants observed from the video recordings were extracted using the coding method Process Coding as described in the book from Saldaña (2013, pp. 264-266).

The three coding methods that were used are described with example codes as follows.

- **In Vivo Coding.** The exact phrase from the participants' answers was extracted into a code. For instance, the code "Easy to navigate a big project and dive deeper" was derived from "*The visualisation is good and it is easy to navigate a big project and dive deeper.*", as written by a respondent.

- **Initial Coding.** The qualitative data in the survey was examined and broken down into discrete parts. For example, the sentence "*I like the zoom-in and out feature and also how the connections between different components are visualized*" was broken into two parts and two codes were extracted from it.

- **Process Coding.** The participant actions and consequences were searched from the video recordings. The code "Viewing commit history" is an example by capturing a participant action.

Afterwards, themes were produced by grouping the codes into coherent categories. By grouping and conceptualising the related themes, theoretical constructs were developed. Finally, a theoretical narrative was created to convey the experience of the research participants with the tasks and the visualisation.

5
Visualisation Tool

The chapter introduces the characteristics and features of the visualisation tool, along with the interaction components.

5.1 Role Stereotype Encoding

Each circle in the graph represents a class in the software system and the colour of the circle represents its role stereotype. The six role stereotypes are encoded by colours as shown in the following table (Table 5.1). A legend for the role stereotypes is placed at the bottom of the visualisation, in which each role stereotype can be hovered over to highlight the corresponding classes, and thus users having problems distinguishing colours are also able to identify the role stereotype of the classes.

Role stereotype	**Colour**
Controller	Purple
Coordinator	Green
Information Holder	Red
Interfacer	Yellow
Service Provider	Blue
Structurer	Pink

Table 5.1: Role stereotype and colour conversion

5.2 Representations of Class, Package, System and Dependency

The concept of a software system is illustrated by a node-link diagram with grey-bordered rectangles representing the packages, and circle nodes representing the classes (Figure 5.1). An arrow (link/edge) illustrates a dependency from a class to the pointing class. For example in Figure 5.2, the class `WalletActivity` depends on the class `Application`. In the context of responsibility-driven design, a class node with body colour indicating the role stereotype represents an object with a specific role, an arrow showing the dependency between two class objects represents the collaboration of the classes to fulfil a responsibility. Chances are high that isolated class nodes without arrows contain static variables and functions, in which no dependencies are created.

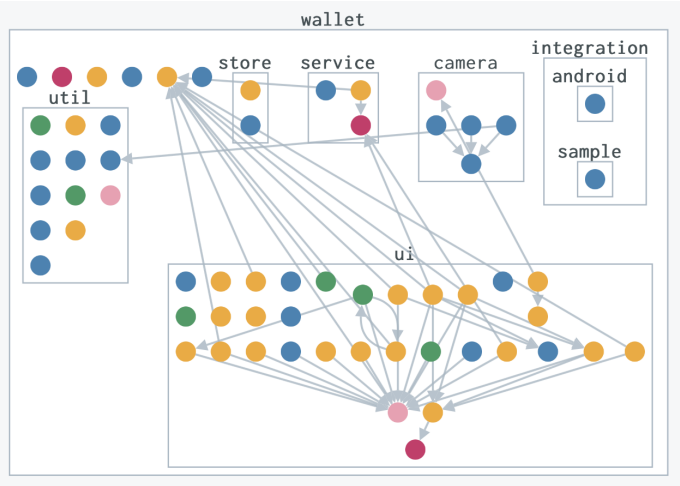


Figure 5.1: Representations of class, package and system in the node-link diagram

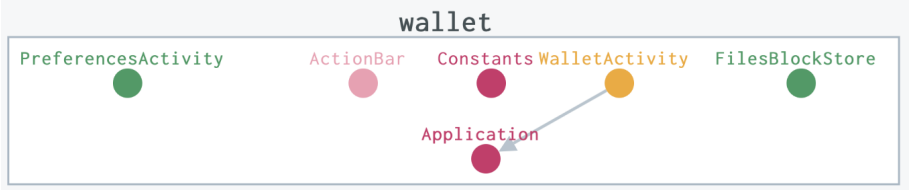


Figure 5.2: Representation for dependencies

5.3 Class Metrics

Two class metrics are incorporated into the visualisation.

Role stereotype. The six role stereotypes are encoded by colours (See Table 5.1).

Lines of code. The size of class nodes is used to encode the number of lines of code for each class, so as to give a rough idea of the differences of classes in terms of lines of code. Figure 5.3 shows the visualisation of the metric lines of code on the class nodes where it is visible that the class `WalletActivity` has the largest number of lines of code while the class `Constants` has the smallest number.

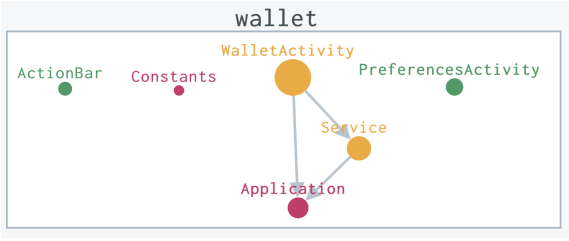


Figure 5.3: Visualising lines of code metric

5.4 Abstraction Levels

A software system is represented by three abstraction levels.

System level. The visualisation shows data of the latest version at the system level by default. All packages and sub packages are displayed in the rendered graph. The sub packages within packages are placed inside the corresponding packages to show the hierarchies. All the classes inside the packages and the class-level dependencies are also shown. Figure 5.1 is an example showing the system level. The system level is accessed by clicking on the grey background outside the parent packages at the package or class level.

Package level. By clicking on a rectangle representing a package, the package level is entered. All the sub packages within the selected package, which is highlighted by the thick black borders, and the classes within both the selected package and its sub packages are shown, as well as the classes in other packages which have dependencies with the classes in the selected package. Take the selected package `wallet.util` as an example (Figure 5.4), all the classes belonging to it are displayed, as well as the class `AbstractWalletActivity` in package `wallet` which depends on the class `ActionBarFragment` in the selected package.

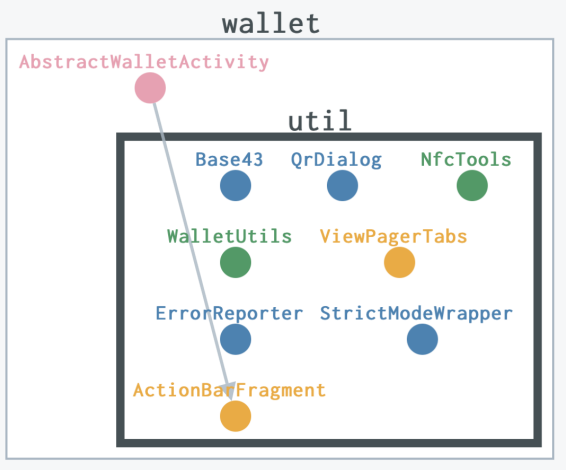


Figure 5.4: Graph at the package level

Package-level dependencies are noticeable at this level. The class-level dependencies across packages reveal how packages utilise each other. For instance, Figure 5.5 shows the package-level dependencies between the selected package `wallet.util` and the packages `wallet`, `wallet.camera` and `wallet.ui`.

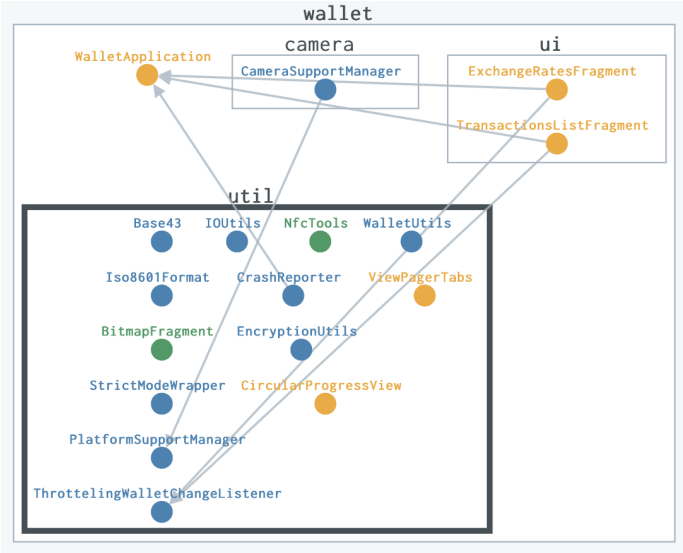


Figure 5.5: Package-level dependencies

Class level. By clicking on any circle class node at the system or package level, the class-level visualisation is displayed. The dependencies between the selected class and classes in all packages are shown. For example, Figure 5.6 shows the selected class `ActionBarFragment` highlighted by a black border at the class level with the class `AbstractWalletActivity` in the package `wallet` depends on it.

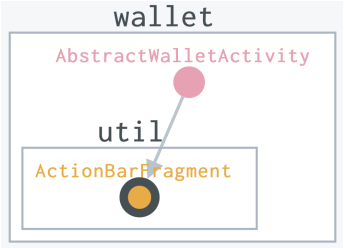


Figure 5.6: Graph at the class level

5.5 Class Dependencies

The inclusion of dependency types and levels is intended to demonstrate how the classes are related to each other within a software system and how a class is utilised by other classes. The latent connections between the classes may be discovered. The options of dependency type and level are only available at the class level. Three options are provided for the dependency type: “All”, “In” and “Out”. The supported options for the dependency level are 1, 2 and 3.

The dependency level is set to 1 by default and thereby the graph only shows the directly connected classes to the selected class. Figure 5.7 shows both incoming and outgoing dependencies for the selected class `PeerListViewModel` within the first dependency level.

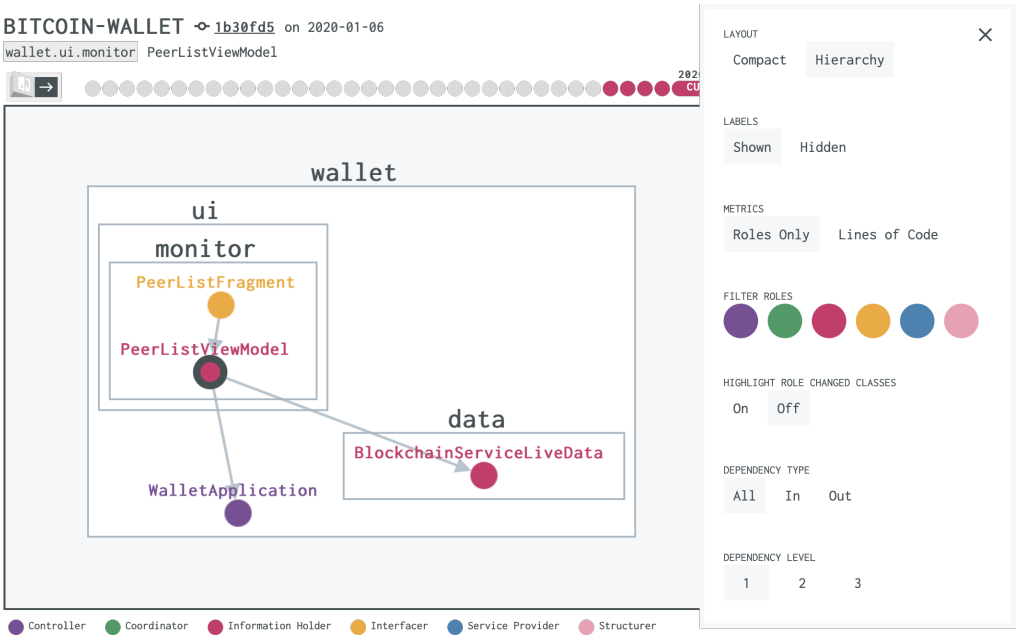


Figure 5.7: Visualisation with the default values for dependency type and level

"All" is set by default for the dependency type, and thereby both incoming and outgoing dependencies are displayed. An incoming dependency implies the connecting class depends on the pointing class while an outgoing dependency has the opposite meaning. Figure 5.7 above shows the selected class `PeerListViewModel` has an incoming dependency from `PeerListFragment` and outgoing dependencies to `WalletApplication` and `BlockchainServiceLiveData`.

When the dependency level is set to 2 or 3, the incomers to the selected class and the incomers of incomers, and so on, are shown for the option "In". Similarly, outgoers for the option of "Out". The dependencies at different levels are represented by the width of the arrows. The thickest arrows illustrate the first-level dependencies, while the medium-sized and thinnest arrows represent the dependencies at the second and third level respectively.

Figure 5.8 demonstrates the dependencies for the class `PeerListViewModel` with only outgoing dependencies within three levels. To simplify the results and reduce the number of dependencies, some classes are filtered. As mentioned previously, the class has outgoing dependencies with `WalletApplication` and `BlockchainServiceLiveData` within the first level, and those first-level dependencies are shown with the thickest arrows. As can be seen in the graph, the arrows representing the dependencies from `WalletApplication` to `Event`, `BlockchainState` and `Configuration` are of medium-sized. This is an indication of second-level dependencies as they are related to the class `WalletApplication`, which is within the first dependency level of the selected class `PeerListViewModel`. The thinnest arrow from `Configuration` to `ExchangeRate` shows the third-level dependency.

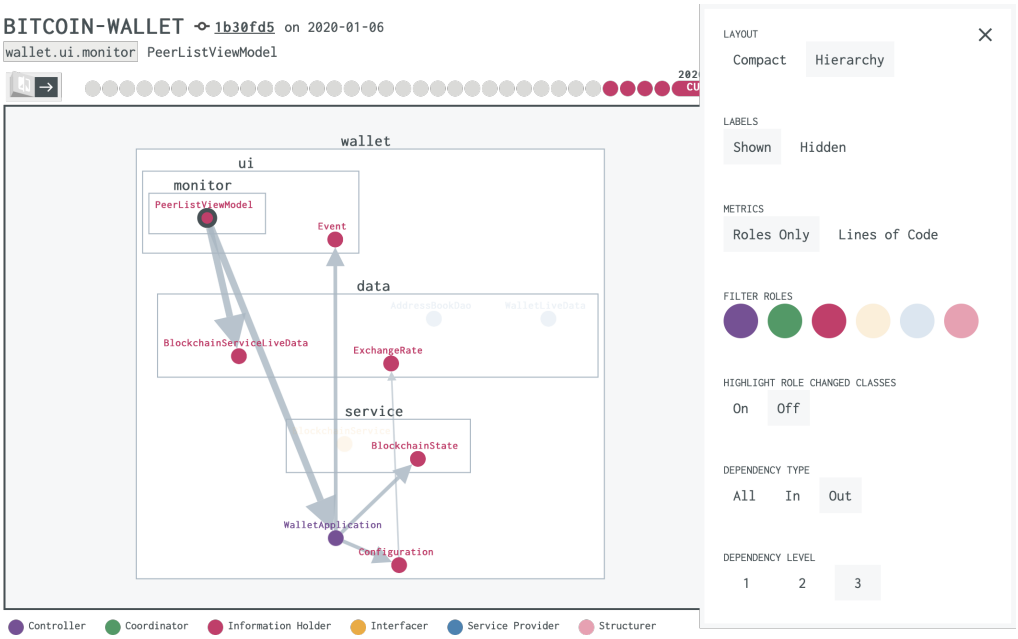


Figure 5.8: Visualisation showing the outgoing dependencies with three levels

5.6 Timeline Constitution

A timeline is created in the visualisation to show the changes of the class role stereotype at the class level, and the changes of the dominant role stereotypes at the system and package levels. Figure 5.9 shows the timeline with different shapes at the three levels.

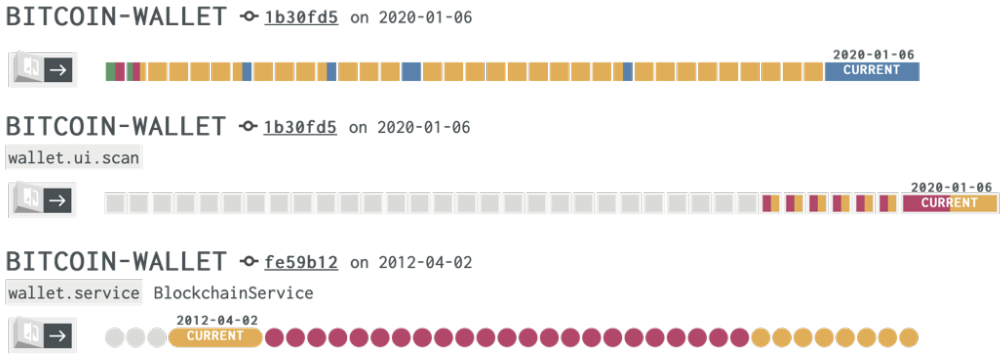


Figure 5.9: Timeline constitution at the system level (top), package level (middle) and class level (bottom)

Each commit version of a software system is illustrated by a coloured shape at different levels. The timeline is put together by a series of shapes with one for each version. At the system and package levels, the colour of the square implies the dominant role stereotypes, in other words, the role stereotype with the highest number of occurrences, in the system or in the selected package respectively. Multiple colours in the square indicate there are more than one role stereotypes having the highest number of occurrences. Grey colour indicates the inexistence of the selected package

in the corresponding version. For example, the graph in Figure 5.10 shows the role stereotypes Information Holder (red) and Interfacer (yellow) are of equally largest number of occurrences in the selected package `wallet.ui.scan`; thus, the shape on the timeline representing the current version has the corresponding two colours (red and yellow). Moreover, the figure shows the inexistence of the package during the first 28 versions.

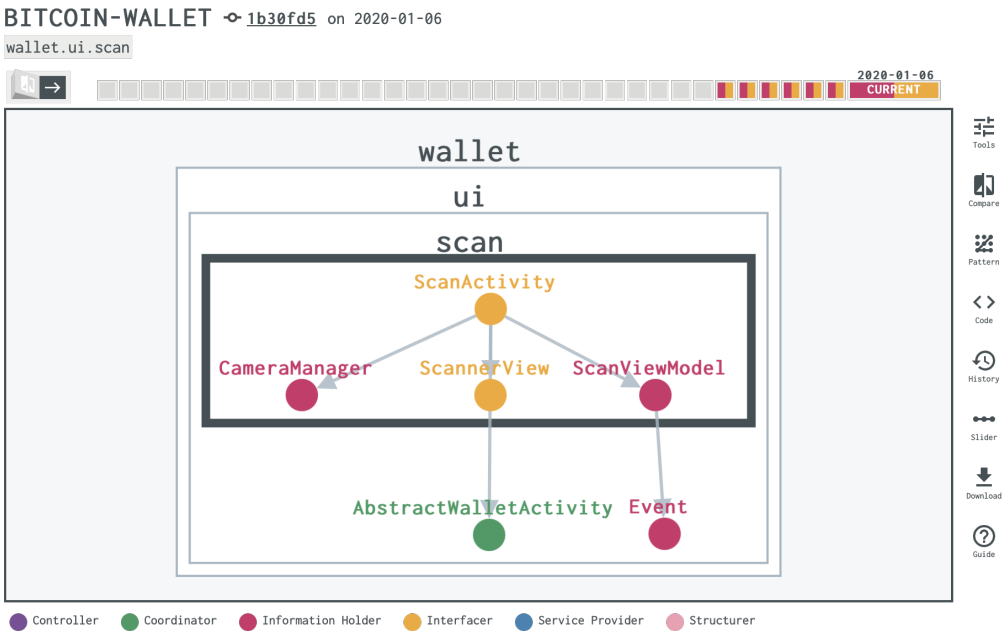


Figure 5.10: Visualisation at the package level

At the class level, the timeline is formed by the coloured circles representing the class with certain role stereotypes in different versions. The grey colour depicts the absence of existence of the selected class in the corresponding version. Figure 5.11 shows an example of the class `BlockchainService`. The first three grey circles on the timeline indicate that the class does not exist in the first three versions. On 2012-04-02, it is added and has the role stereotype Interfacer, then becomes Information Holder for 23 versions. After that, it becomes Interfacer again for 8 versions. In this way, the timeline serves as an indication of how and when the changes of role stereotype of the selected class occur.

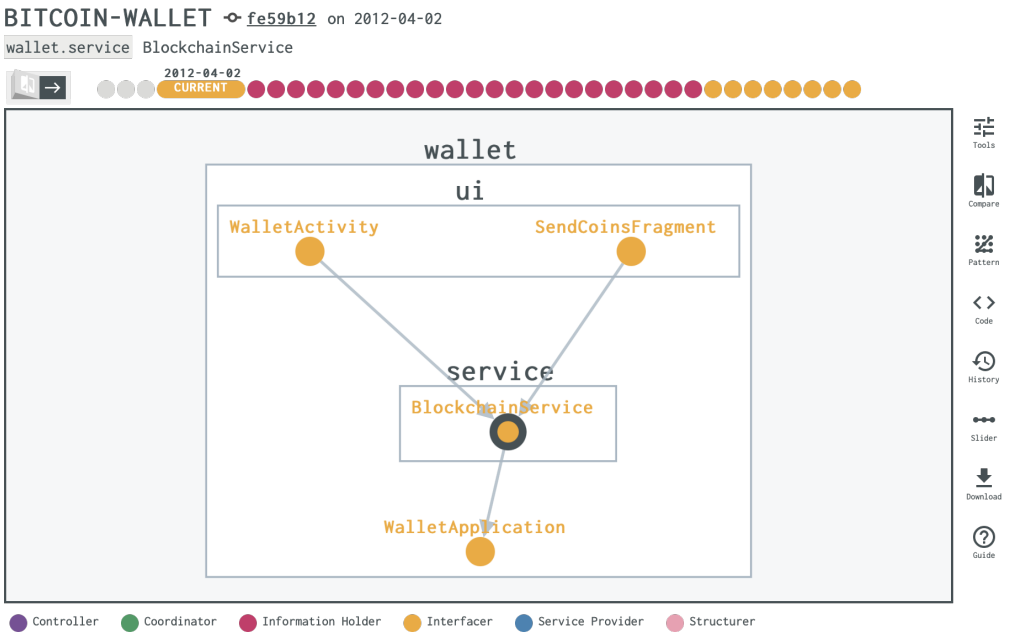


Figure 5.11: Visualisation at the class level

The current viewing version is annotated on the timeline, and it can be switched by selecting a shape representing another version on the timeline. Moreover, by switching to the comparison mode using the toggle button on the left, the comparing version can be chosen on the timeline. Figure 5.12 shows the toggle button with the comparison mode and the annotations of the current and comparing versions on the timeline.

Figure 5.12: Timeline with current and comparing versions annotated

5.7 Pairwise Comparison

The visualisation enables pairwise comparison of two selected versions at any of the three levels: system, package and class. In order to do the pairwise comparison, one must select a version to compare. There are two ways to select a comparing version. First, the dropdown select box in the compare dialog can be used. Second, the toggle button with comparison mode next to the timeline can be used to select a version for comparison within the timeline.

After selecting the version to compare, the changes between the current viewing version and the selected comparing version are displayed immediately. The differences between them are displayed as changes from the earlier version to the later version, no matter if one is comparing the current version to an earlier or a later version. Figure 5.13 illustrates the changes between two versions at the system level. The

changes of class are grouped into four types, i.e. added, unchanged, removed and role-changed, while the changes of dependency are grouped into added, unchanged and removed. The word unchanged means the role stereotype of the class remains the same, and the class or dependency exists in both versions, and this is represented by a circle or an arrow with faded colour.

A role-changed class, which is illustrated by a coloured circle with a border of a different colour, implies that the role stereotype of the class in the current version (body colour) is different from that in the comparing version (border colour). An added class or dependency is denoted by a circle or an arrow with solid colour, so as to distinguish with unchanged classes and dependencies. The grey colour on the circle indicates a class is removed in the current version while a dashed arrow line represents a removed dependency. A legend of the notations used for the changes is placed inside the compare dialog.

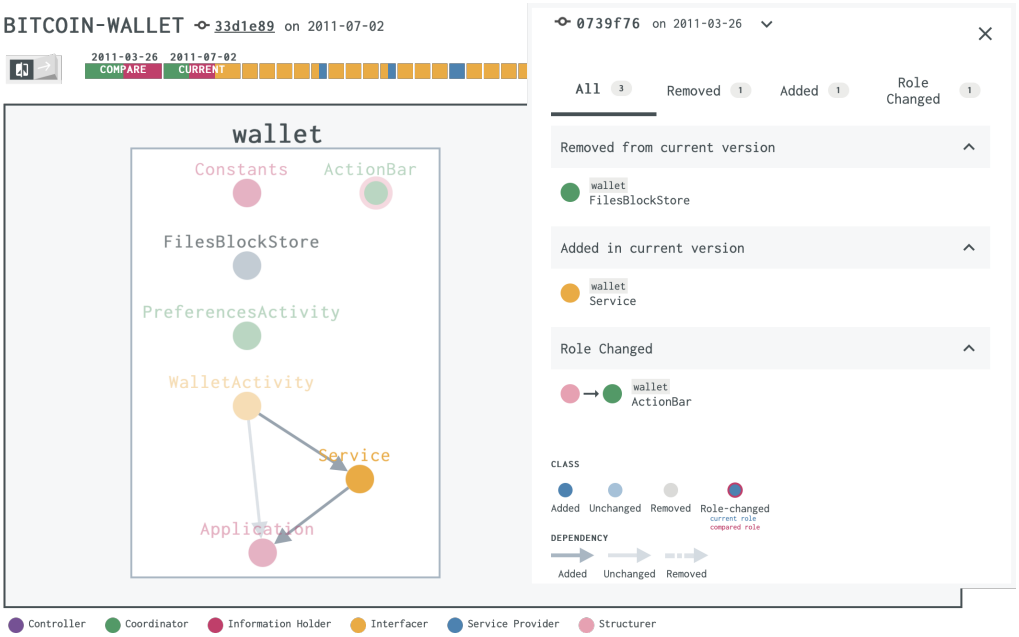


Figure 5.13: Pairwise comparison at the system level

In the example shown in Figure 5.13, the versions on 2011-03-26 and 2011-07-02 are compared. The class `FilesBlockStore` in the package `wallet` is removed while the class `Service` is added in the version on 2011-07-02. Dependencies from the class `WalletActivity` to the class `Service` and from `Service` to `Application` are added. The class `ActionBar` having the colour green in the body and pink in the border indicates that the role stereotype is changed from Structurer to Coordinator, as the current version is 2011-07-02. The classes `Constants`, `PreferencesActivity`, `WalletActivity` and `Application`, and the dependency from `WalletActivity` to `Application` are unchanged between the two versions.

The representations of different types of changes are the same for both the package and class levels. However, one has to be aware that the meaning of added and removed classes at the package and class levels is different from that at the system

level. They simply imply the existence or nonexistence of classes in the system at the system level. On the other hand, the addition or removal of classes in the comparison at the package and class levels means the related classes have added or removed dependency from or to the selected package or class. In other words, the meaning of added or removed classes at these two levels can be interpreted as linked or unlinked classes respectively, or added or removed collaborating classes in terms of collaborations. A removed collaborating class may or may not be removed from the system. However, chances are high that a removed class at the class or package level is removed from the system.

As can be seen in Figure 5.14, the dependency from the class `AbstractWallet-Activity` to `ActionBarFragment` is removed in the version on 2012-04-02. The class `AbstractWalletActivity` is considered as a removed class in connection to the selected package `wallet.util`. The visualisation does not keep track of relocation of classes to another package; however, this may be noticed by addition and removal of classes with the same name. For example, `AbstractWalletActivity` is removed from `wallet` and added in the package `wallet.ui`. Likewise, rename of a class may be recognised in the same way with similar names.

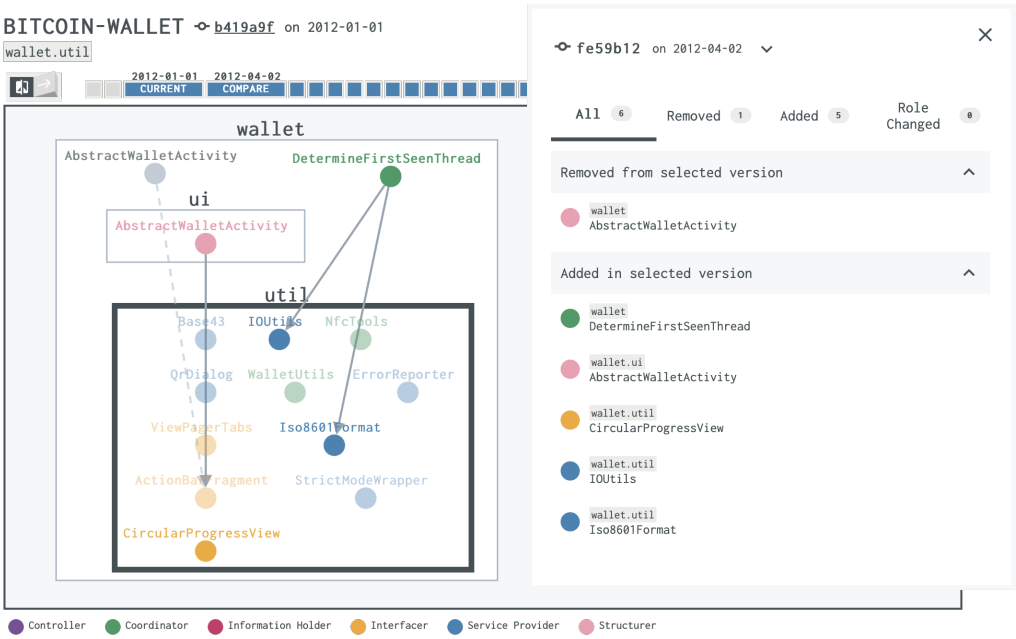


Figure 5.14: Pairwise comparison at the package level

The visualisation also provides code comparison of the selected class between two versions with highlighted differences, with the source code for the earlier version on the left. An example for the class `BlockListViewModel` can be seen in Figure 5.15.

Figure 5.15: Comparison of source code in two versions

Combining the information of changes of role stereotype on the timeline and the changes of collaborating classes shown in the diagram, one may trace where the responsibilities of the selected role-changing class shift to, as a change of role stereotype probably indicate the responsibilities of the class is redistributed to its collaborating classes. For example, Figure 5.16 shows the added collaborating class `BlockchainServiceImpl` and removed dependencies from `WalletActivity` and to `WalletApplication` for the selected class `BlockchainService` when its role stereotype changed from Interfacer to Information Holder, which is probably an indication of the responsibilities of `BlockchainService` are transferred to its new collaborating class `BlockchainServiceImpl` (Interfacer).

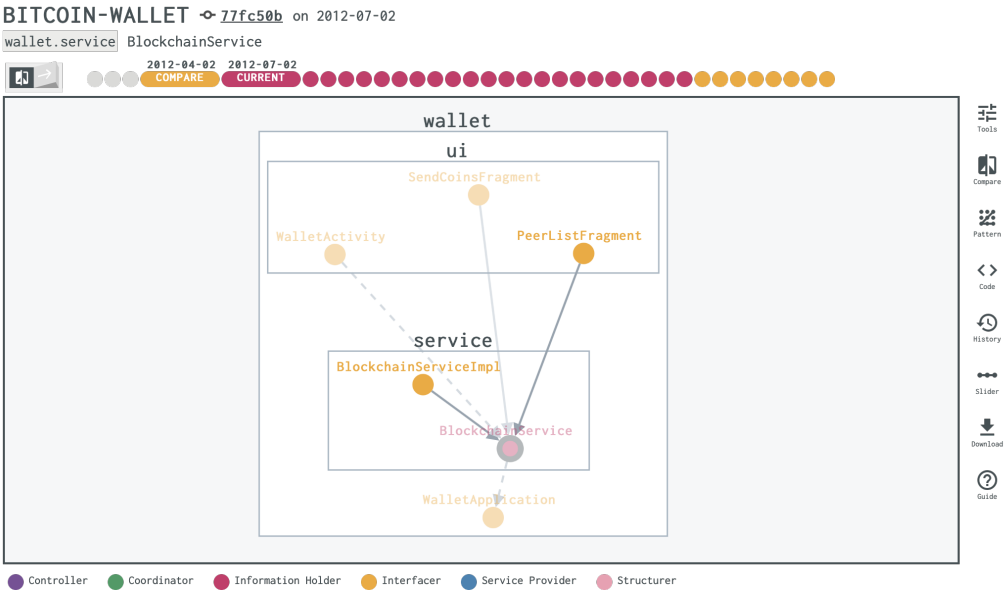


Figure 5.16: Combination of changes of role stereotype and collaborating classes at the class level

5.8 Consecutive Comparison

A set of slider control and previous and next buttons is included for showing the changes between two consecutive commit versions each time. By selecting a version with the slider control, the graph displays the changes between the selected version and one version before it. Alternatively, the previous and next buttons can be used to select a version. Continuously clicking on the next button allows users to view the changes chronologically for the entire range of versions. The representations for added, removed and unchanged classes and dependencies as well as the role-changed classes in this view are the same as those in the comparison of two versions. Figure 5.17 shows the changes centred the package `wallet.util` between the selected version on 2012-04-02 and the previous version, for example, the dependency from the class `AbstractWalletActivity` to `ActionBarFragment` is removed and the classes `IOUtils` and `Iso8601Format` are added.

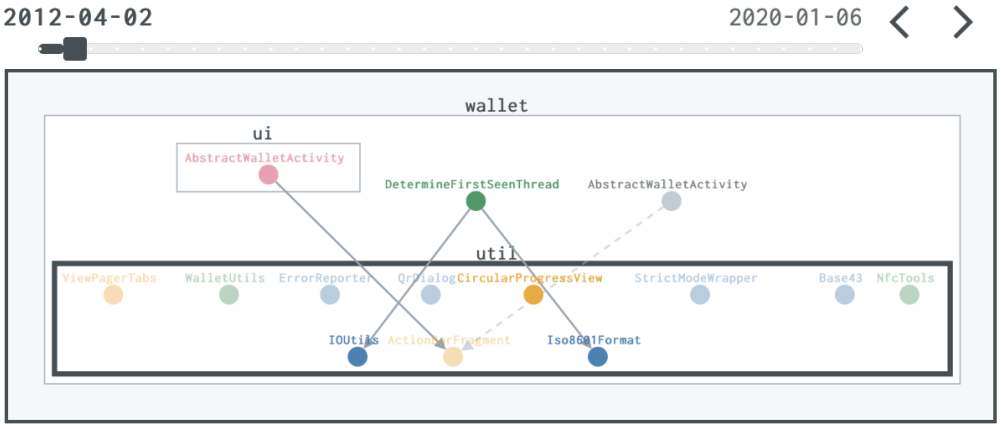


Figure 5.17: Slider and changes between two versions

5.9 Pattern Finder

The visualisation enables searching for specific collaboration patterns within the system. A collaboration pattern is the collaborations between the objects within different dependency levels. The pattern finder provides three ways to look for a specific collaboration pattern: user-defined pattern, ranking lists and common patterns. It supports two levels of collaboration patterns, in which at most three classes are involved in a case matching the pattern.

The feature of user-defined pattern is created for users who are knowledgeable on role stereotypes and know what they are looking for. When only the role stereotypes for the first class are selected, the classes with the selected role stereotypes are displayed and their total counts are shown in the chart, which illustrates the changes of the occurrences over the versions. For example, Figure 5.18 shows the role stereotypes Controller and Coordinator selected for the first class; thus, the graph displays only the classes with the selected role stereotypes. The line chart shows the total

numbers of the selected role stereotypes over the versions, with 12 counts in the current version.

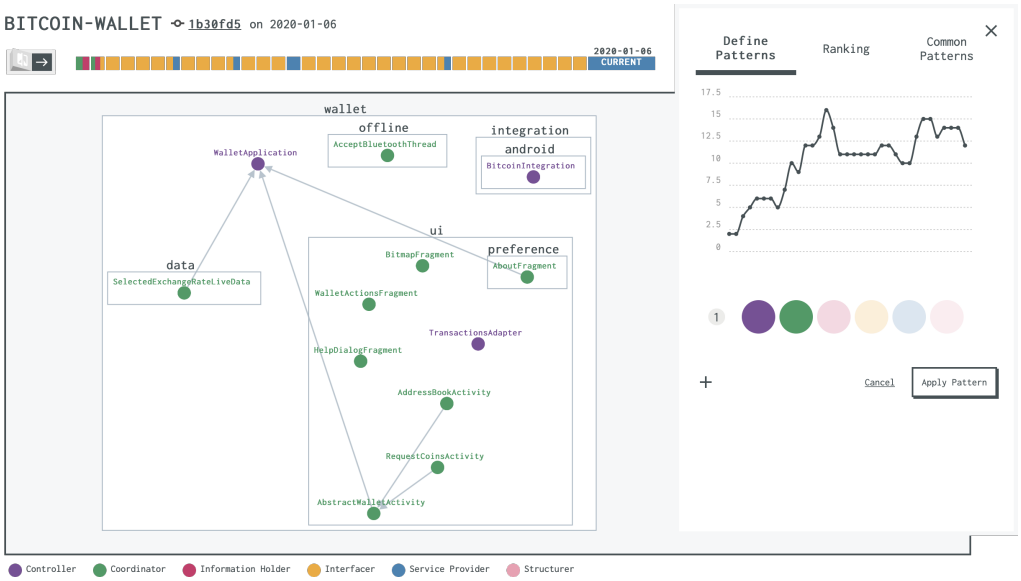


Figure 5.18: Results of the user-defined pattern for the first class

For user-defined patterns with one or two levels, the visualisation displays the result of cases matching the selected role stereotypes at the levels. For example, Figure 5.19 shows the results of the defined pattern Interfacer to Structurer to Information Holder. Three results matching the collaboration pattern are found. One of them is the class `ExchangeRatesFragment` (Interfacer) depending on `ExchangeRatesAdapter` (Structurer), of which depends on `BlockchainState` (Information Holder).

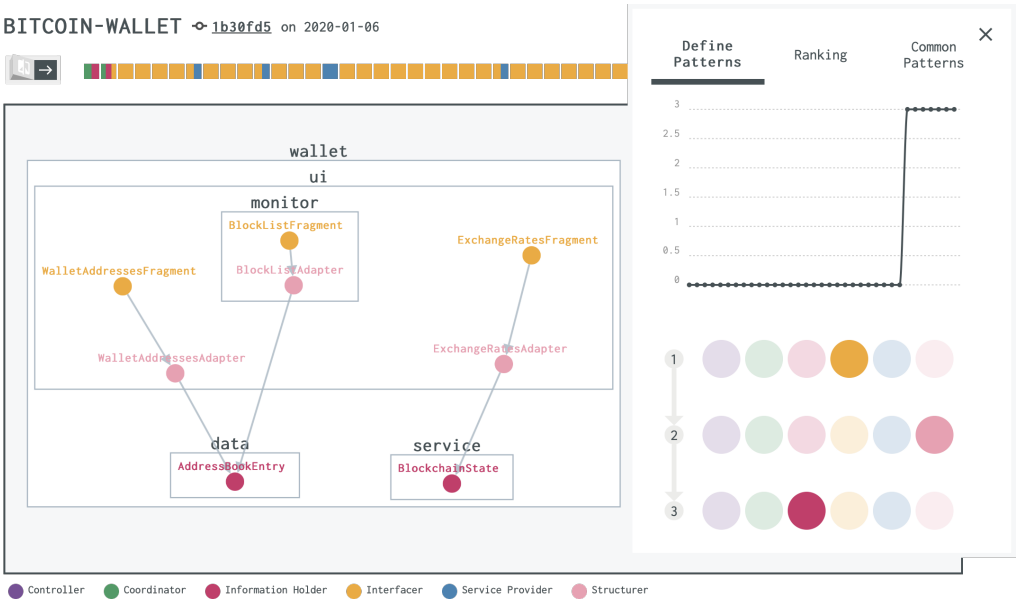


Figure 5.19: Results of the user-defined pattern for two levels

The visualisation also identifies all possible collaboration patterns within two levels and adds up the number of occurrences to generate a ranking of all the patterns existing in the current version. After selecting a pattern in the list, the diagram shows all the cases matching the selected collaboration pattern, and its changes of the number of occurrences are shown in the line chart. For example, Figure 5.20 shows parts of the ranking list on the right and the results of the selected pattern, i.e. from Interfacer to Information Holder, on the left.

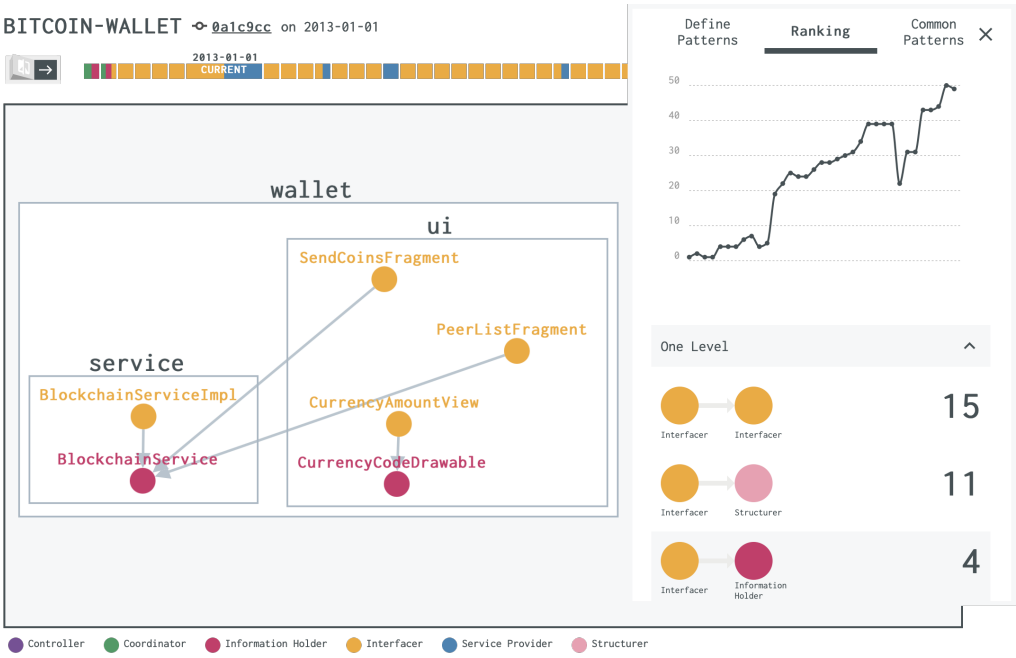


Figure 5.20: Results of the selected pattern in pattern ranking list

In addition to the user-defined pattern and ranking list of all available patterns, some common patterns are suggested for users who are unfamiliar with role stereotypes and have no ideas what patterns to look for. Figure 5.21 shows the results of a common pattern (Coordinator to Interfacer to Service Provider).

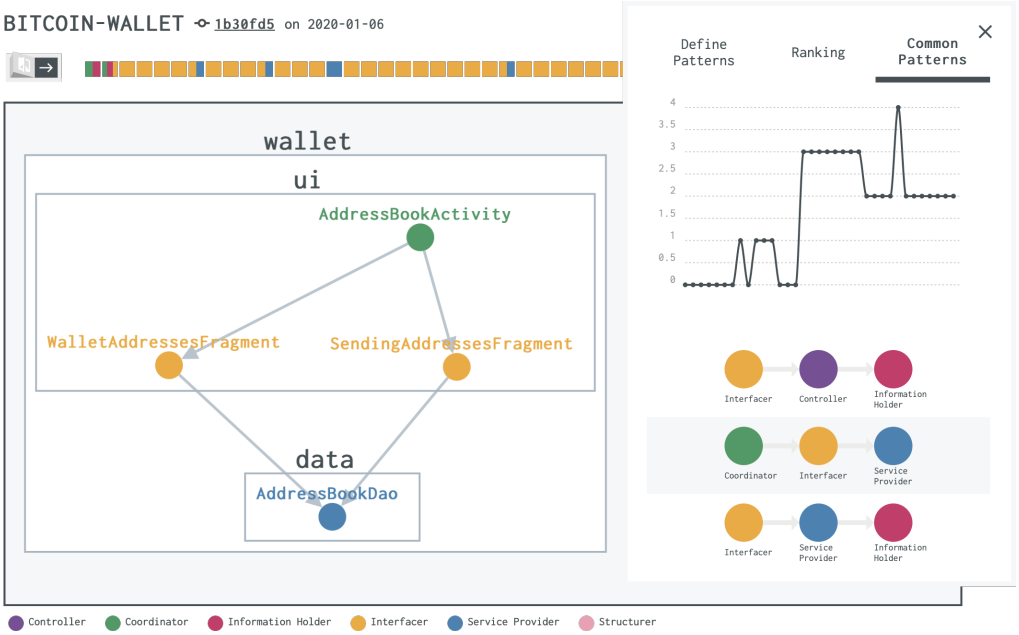


Figure 5.21: Results of the selected common pattern

5.10 Role Stereotype Changed Indication

The visualisation provides a feature to highlight all the classes that have changed role stereotypes over all versions. Figure 5.22 shows the highlighted class `WalletUtils` in the graph, indicating that the role stereotype of the class has changed over the versions, while that of other classes without a dashed border have never changed.

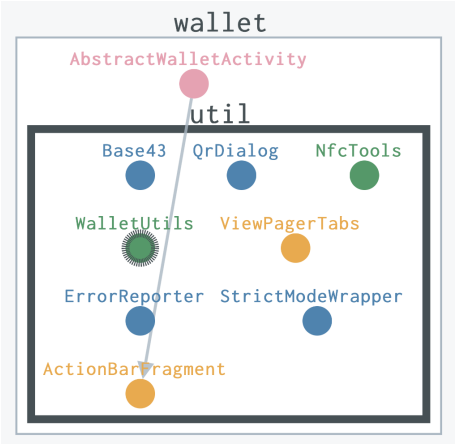


Figure 5.22: Highlighting classes which have changed in role stereotype

5.11 Miscellaneous

The visualisation also provides the following features.

History. When an element in the diagram is selected, a record holding the current version and the selected element is stored in the history list. The viewed elements (system, package or class) are listed, and the list can be exported to a text file. It is enabled to go back to previously viewed elements by clicking an item in the history list. It is also possible to clear the history list.

Graph export. The rendered graph can be downloaded in the format of Portable Network Graphic (PNG).

Source code. The source code for classes is displayed within the visualisation for better comprehension between the visualisation and the actual code without having to look for it. The syntax is highlighted with the styles used in Eclipse, a widely used Java integrated development environment (IDE).

5.12 Interaction Components

The visualisation design has adopted all of the seven categories of interaction techniques proposed by Yi et al. (2007).

Select. Users can select a specific class or package by clicking a node or a rectangle respectively. The node or rectangle is highlighted with a black border around it; therefore, users can keep track of the class or package easily after applying different features.

Explore. Users can explore the data at the system level by default, or subset of data by clicking a node or rectangle to view the data at class or package level respectively.

Reconfigure. The visualisation enables users to choose between different types of layout, currently compact and hierarchy layouts are available.

Encode. The class nodes are by default of the same size, showing the role stereotype only. Moreover, alternative representation is provided by altering the size based on the encoded number of lines of code for the classes.

Abstract/Elaborate. When the class names are set to not shown, users can hover over a class node to view its name. Moreover, zooming and dragging are possible for all elements in the graph. Users can alter the scale of the graph to look at the detailed view or the overview of the dataset by zoom-in or zoom-out respectively.

Filter. Users can alter the graph by filtering different role stereotypes. Figure 5.23 shows the graph rendered after filtering out the classes with the role stereotypes

Information Holder and Interfacer. The filtered classes are in solid colours while those filtered out are in faded colours.

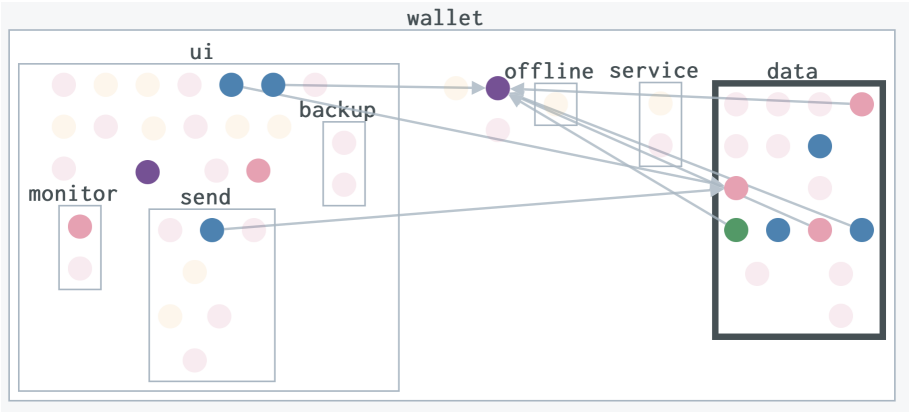


Figure 5.23: Visualisation with Information Holder and Interfacer filtered out

Connect. The directly connected classes and their neighbours within all dependency levels are highlighted when hovering over a specific class node. When hovering over the circle representing a role stereotype in the legend, the classes having the corresponding role stereotype are highlighted. Moreover, the graph highlights the package and all collaborating classes when hovering over a package.

5.13 Implementation

The visualisation tool is implemented as a web application, using the HTML templating engine Pug, SCSS and JavaScript for front-end development. Express and Node.js are adopted as the web framework and the runtime environment respectively. The data for the software system to be visualised in the artefact is stored in the format of comma-separated values file (CSV), with an alternative option of connection to the graph database Neo4j. The graph visualisation is implemented with the use of the JavaScript graph theory library Cytoscape.js and the Klay layout algorithm for Cytoscape.js. CodeMirror is used to show the source code and highlight code changes.

The tool currently requires data inputs of role stereotypes and dependencies. The visualisation tool can basically be applied to any object-oriented software systems, provided that it is integrated with the role stereotype classifying tool (Nurwidyantoro et al., 2019; Ho-Quang et al., n.d.) and modified to achieve automated extraction of dependencies.

6
Results

This chapter presents the results of implementing the methodology described in Chapter 4.

6.1 Design Science Research

The final artefact produced, i.e. the visualisation tool named Rologram, of which the name is derived from "role evolution diagram", using the design science research approach is introduced in Chapter 5. The implementation of the visualisation has gone through three iterations of design cycle. The results from the three iterations of development and evaluation are described as follow.

6.1.1 First Iteration

During the design and development stage of the first iteration, the component consecutive comparison was implemented. Figure 6.1 shows the screenshot of the visualisation built. The visualisation provided simple features of showing the changes between two consecutive versions by using the slider on top to select a version and a legend for mapping the colours and role stereotypes.

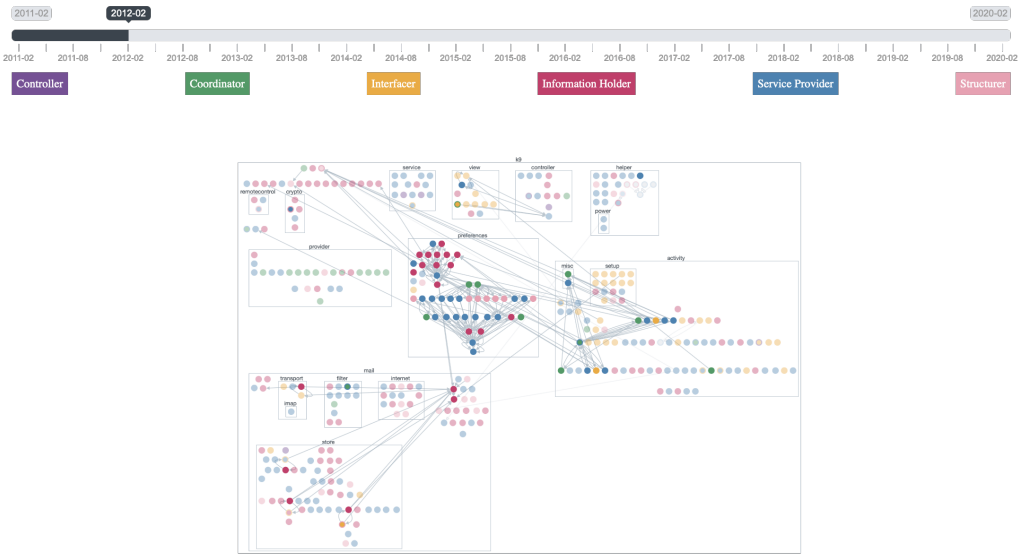


Figure 6.1: Screenshot of the visualisation for the evaluation in the first iteration

After the design and development in the first iteration, evaluation on the visualisation tool was conducted with three experts in software evolution and class role stereotypes.

They pointed out the following issues with the visualisation:

- Navigation between the versions
- Lack of legend of the representations for changes, in particular the circles with a different border colour
- Popup of class name
- No highlight of selected class and package

Moreover, they suggested extra features including:

- Viewing of source code for the classes
- Addition of class metrics, for example, lines of code
- Filtering of role stereotypes
- User guide for the visualisation

6.1.2 Second Iteration

After the first iteration of the design cycle, the visualisation was refined based on the feedback from the first evaluation, and all feedback was considered and incorporated into the visualisation, except the user guide was not fully completed. As a result of the design and development activity, the functionalities of pairwise comparison, pattern finder, display of code, history and options for changing the layout and metric were implemented. Figure 6.2 shows a screenshot of the artefact, with the data of K-9 Mail system.

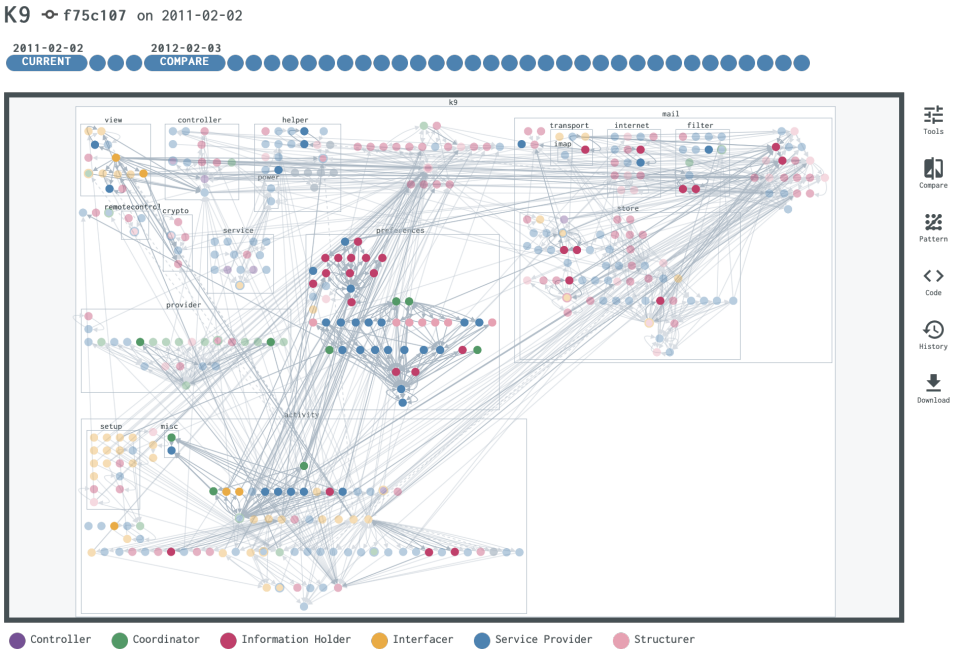


Figure 6.2: Screenshot of the visualisation for the evaluation in the second iteration

Two experts from the first evaluation and one other participated in the evaluation of the built artefact during the second iteration with the following comments (Table 6.1).

Acknowledgment	Suggestions
1. nice representation of the lines of code metric for the classes 2. nice pattern builder (the user-defined pattern) 3. chart showing the evolution of the number of occurrences of the defined pattern 4. fast and smooth visualisation	1. missing legend 2. selected layout should be preserved 3. shape used on the timeline for the system and package levels is the same as that for the class level which causes confusion 4. filtered classes can be faded and remained in the graph 5. unable to know all available functionalities 6. some circles are overly large when visualising the lines of code metric 7. want dependencies to other packages at the package level

Table 6.1: Comments from the second evaluation

6.1.3 Third Iteration

With consideration of all the feedback from the evaluations in the first two iterations, the final version of the artefact is created. Figure 6.3 shows the screenshot of the artefact for the evaluation in the third iteration, with the data of Bitcoin Wallet system.

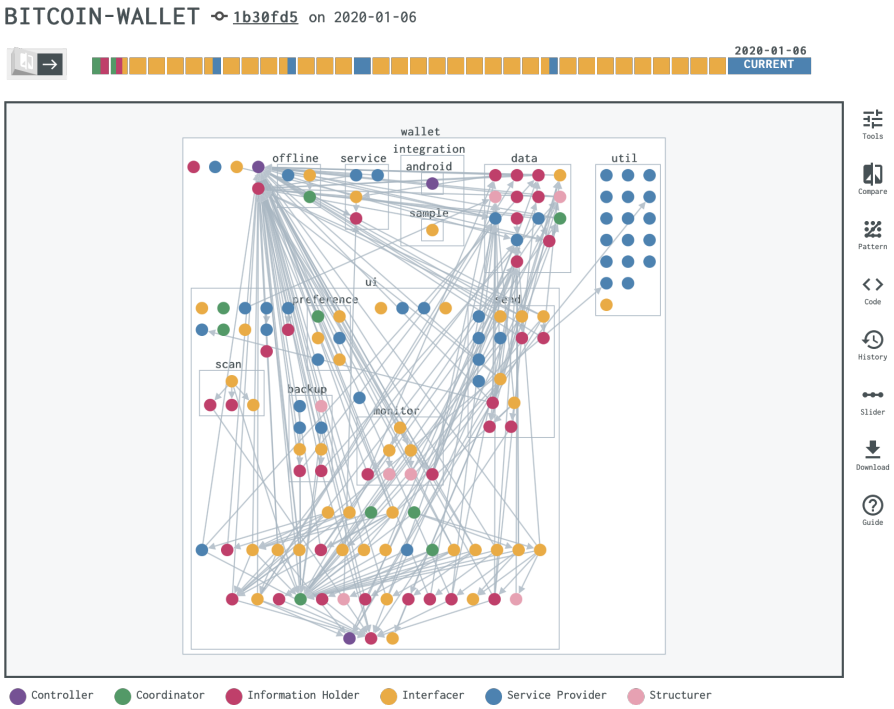


Figure 6.3: Screenshot of the visualisation for the final evaluation

The user study was conducted as the final evaluation of the design science research and the feedback collected from the participants are shown in Table 6.2.

Acknowledgment	Suggestions
1. easy to go back with history function 2. easy to navigate a big project 3. easy to use 4. visualising the content of different commits is fast 5. display of the code for selected class is fast 6. efficient tool 7. friendly user interface 8. zoom-in and out feature 9. visualisation of the connections between different components 10. help finding dependencies across packages 11. useful representation of metric lines of code 12. highlighted newly added code	1. accidental double click causes misbehaviour 2. too many lines prevent from clicking the package/class 3. often jump to other pages or refresh unwantedly 4. removed dependency should be in bold colour 5. history does not keep track of the comparing version 6. lack of animation to display the changes 7. user guide should pop up after initialisation 8. missing definition of role stereotypes 9. dependencies should not be preloaded when entering the page/include an option to change the visibility of dependencies 10. unable to hide some versions 11. use of pink colour causes confusion with the dimmed red colour 12. unclear meaning of icons on toggle button 13. unable to filter the versions 14. indicators such as year on the timeline

Table 6.2: Comments on the visualisation from the final evaluation

6.2 User Study

6.2.1 Demography

The participants for the user study were in various occupations, including product owner, postdoctoral researchers in software engineering, back-end developer, software engineer and IT placement. All respondents have never used nor contributed to the software system Bitcoin Wallet involved in the tasks, although one of them has studied the source code. The demographic characteristics of the participants are summarised in the charts below. The pie charts in Figure 6.4 show the distribution of gender and age of the participants.

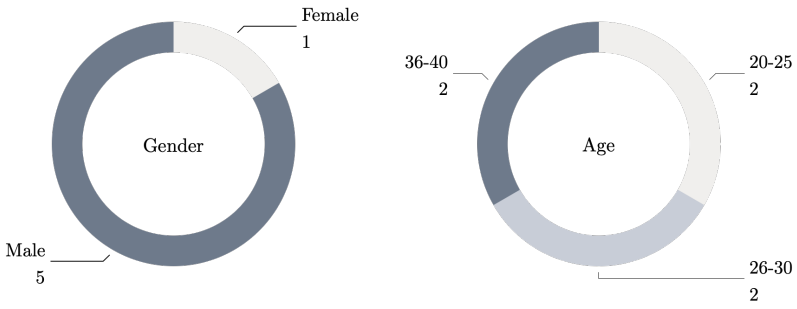


Figure 6.4: Distribution of gender (left) and age (right)

Figure 6.5 shows the distribution of the Java experience, programming experience and work experience in the Information Technology (IT) industry of the participants. The range of participants covered varying experience in terms of programming and work experience, and yet no participants had Java programming experience for longer than five years.

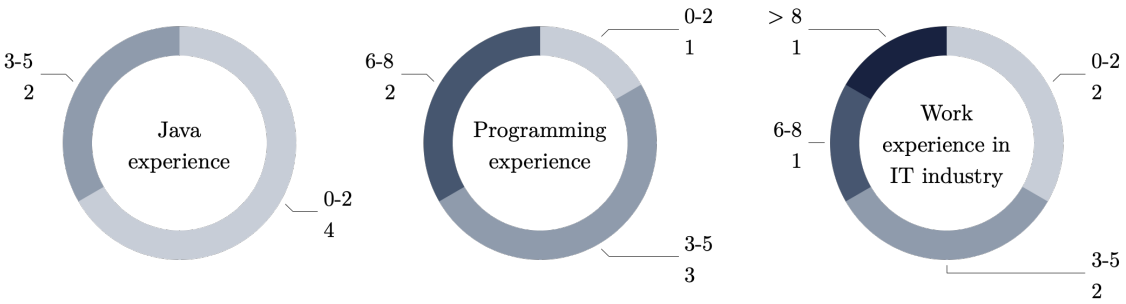


Figure 6.5: Distribution of Java (left), programming (middle) and work experience (right)

The participants were asked whether they have experience in the following areas: software development, software maintenance, software testing, refactoring, debugging, Android development and iOS development. Figure 6.6 shows the numbers of participants having experience in each of the mentioned areas. As can be seen from the figure, the majority of the respondents had experience in software development, followed by software maintenance and Android development.

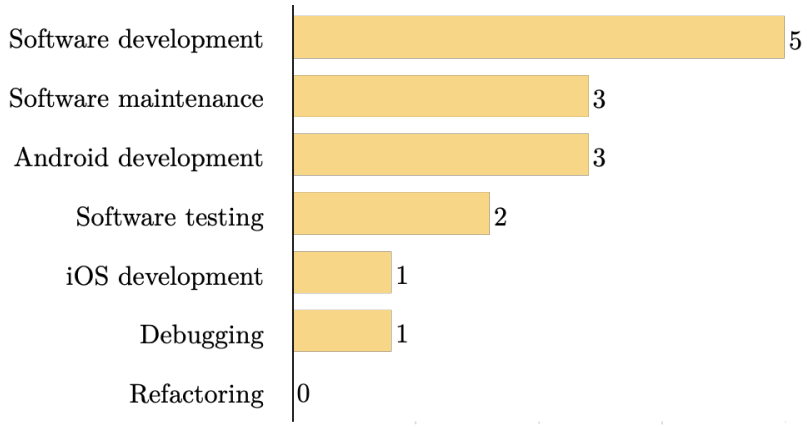


Figure 6.6: Areas the participants had experience in

6.2.2 Task Performance

The combinations and ordering of the tasks and tools for the participants are listed in the following table (Table 6.3). A and B refer to the tasks related to the packages `wallet.service` and `wallet.ui.send` respectively.

Participant	Task 1	Tool for Task 1	Task 2	Tool for Task 2
1	A	with visualisation	B	without visualisation
2	A	without visualisation	B	with visualisation
3	B	with visualisation	A	without visualisation
4	B	without visualisation	A	with visualisation
5	A	with visualisation	B	without visualisation
6	B	with visualisation	A	without visualisation

Table 6.3: Combinations and orderings for the tasks and tools

The task performance of the respondents was measured by the time taken and the points judging from the answers. Table 6.4 lists the total points the participants received in task with the usage of visualisation in comparison to that without, whilst the points for the four task questions are shown in Table 6.6 to 6.9. Table 6.5 shows the time taken for working on the tasks with and without the visualisation tool.

As can be seen in Table 6.4, the participants tended to receive higher points in task with the use of the visualisation compared to that without. Four participants stopped working on the task without the use of the visualisation after 35 minutes, which was the maximum time allowed for working on a task (See Table 6.5).

Participant	With	Without
1	19/20	5/20
2	20/20	7/20
3	18/20	5/20
4	14/20	10/20
5	20/20	7/20
6	10/20	2/20

Table 6.4: Total points for tasks with and without visualisation

Participant	With	Without
1	34:43.82	31:18.73
2	31:59.31	35:01.71
3	20:40.31	35:01.21
4	24:33.09	35:01.07
5	34:48.09	35:01.24
6	29:06.23	30:43.96

Table 6.5: Time taken for tasks with and without visualisation (MM:SS.ss)

As can be seen from Table 6.6 to 6.9, some participants did not manage to answer all the questions in the task, and this is marked in the table with "No answers". Overall, the participants tended to complete all the questions in task with the use of visualisation within the given time, whereas some did not manage to answer some of the questions in task without the use of visualisation. Question 2 regarding changes of package-level dependencies had the greatest number of empty answers, followed by question 4 and question 1.

Participant	With	Without
1	5/5	5/5
2	5/5	No answers
3	5/5	No answers
4	5/5	5/5
5	5/5	0/5
6	0/5	0/5

Table 6.6: Points for question 1 regarding changes of class-level dependencies with and without visualisation

Participant	With	Without
1	4/5	No answers
2	5/5	No answers
3	3/5	No answers
4	4/5	No answers
5	5/5	No answers
6	0/5	0/5

Table 6.7: Points for question 2 regarding changes of package-level dependencies with and without visualisation

Participant	With	Without
1	5/5	No answers
2	5/5	5/5
3	5/5	5/5
4	5/5	5/5
5	5/5	5/5
6	5/5	0/5

Table 6.8: Points for question 3 regarding the responsibility of a class with and without visualisation

Participant	With	Without
1	5/5	No answers
2	5/5	2/5
3	5/5	No answers
4	0/5	No answers
5	5/5	2/5
6	5/5	2/5

Table 6.9: Points for question 4 regarding the changes of responsibility of a class with and without visualisation

6.2.3 Other Quantitative Data

The participants were asked regarding their attitude towards the visualisation after working on the two tasks, on a scale from 1 to 5 (See Figure 6.7).

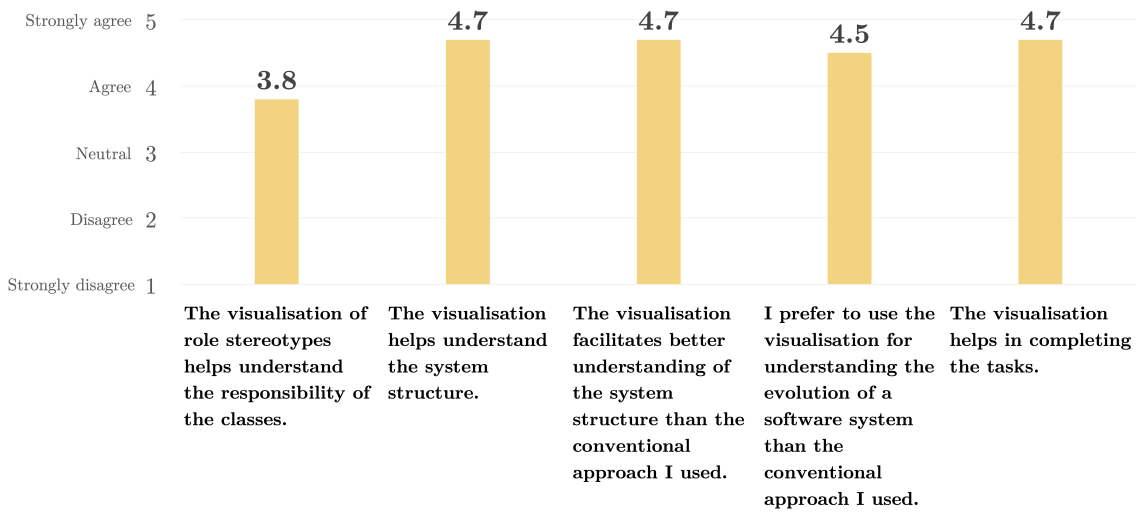


Figure 6.7: Attitude of the participants towards the visualisation

6.2.4 Theoretical Constructs, Themes and Codes

The theoretical constructs (capitalised), themes (italics) and codes extracted from the qualitative data are listed below, with codes extracted by Process Coding from the screen recordings in the format of gerund and those by In Vivo Coding in quotes. The number of occurrences of the corresponding code in survey answers and resulted from Process Coding (observed behaviours and consequences) are included in the parentheses and square brackets respectively.

I. **Approach with the visualisation**
   A. *Accessing abstraction levels*
      1. Viewing the system [26]
      2. Viewing a package [74]
      3. Viewing a class [39]
   B. *Selection of element in the diagram*
      1. Clicking on background to access to system level [10]
      2. Clicking a package (2) [32]
      3. Clicking a class (1) [24]
      4. Hovering over a package [30]
      5. Hovering over a class [77]
      6. Dragging a package [4]
   C. *Operations on timeline*
      1. Hovering over a version on timeline [657]
      2. Selecting the current version on timeline (2) [68]
   D. *Ways to compare two versions with the visualisation*
      1. Clicking toggle button (2) [101]
      2. Using slider (1) [2]
      3. Using compare dialog (2) [27]
   E. *Selection of comparing version*

1. Selecting a comparing version in compare dialog (2) [10]
2. Selecting a comparing version on timeline [83]
3. Selecting a version with slider [7]
4. Using the previous button to select a version on slider [10]
5. Using the next button to select a version on slider [34]

F. *Change identification with the visualisation*
1. Viewing changes of system [1]
2. Viewing changes of package [74]
3. Viewing changes of class [18]
4. Viewing added class list [1]
5. Viewing removed class list [1]
6. Reading change legend (1) [6]

G. *Dependency identification with the visualisation*
1. Identify package-level dependency with arrows (3)
2. Identify added/removed dependencies between classes (1)
3. Identify class-level dependencies (1)

H. *Role stereotype utilisation*
1. Identify role stereotype with colour (3)
2. Spot the change in role stereotype with the colours (2)
3. Identify significant change of role stereotype with timeline (3)

I. *Codes in the visualisation*
1. Reading code in visualisation (3) [10]
2. Viewing code comparison in visualisation [8]
3. Highlighting code in visualisation [4]
4. "Read highlighted codes to observe the change and new functionality" (1)
5. Searching a keyword in source code [1]

J. *Supporting features*
1. Going back to previously viewed element using history feature [5]
2. Hovering over role stereotype legend [1]
3. Downloading the graph as image [1]
4. Zooming in (1) [26]
5. Zooming out (1) [19]

K. *Unexpected behaviours*
1. Double clicking on a package [7]
2. Double clicking on a class [2]
3. Double clicking on one version on timeline [1]
4. Clicking on an edge [21]
5. Opening the code dialog at package level [2]

II. **Approach without the visualisation**

A. *Code retrieval*
1. Cloning code repository using GitHub Desktop (1) [2]
2. Downloading code from GitHub website (1) [2]
3. Opening code repository on GitHub website (3) [33]

B. *Utilisation of tools*

1. Using Eclipse (1) [1]
2. Using Visual Studio Code (1) [1]
3. Using Sublime Text [1]
4. Using Sourcetree [1]
5. Using command line [1]
6. Using GitHub Desktop [2]
7. Using GitHub website (2) [5]

C. *Navigation*
1. Clicking an untargeted package folder [20]
2. Clicking a targeted package folder (2) [106]
3. Hovering over an untargeted package folder [47]
4. Hovering over an untargeted class [25]
5. Clicking a class to open code (3) [129]
6. Going to previous page [23]
7. Navigating to another class from code [5]
8. Switching tabs (1) [552]

D. *Code viewing*
1. Reading code (6) [65]
2. Reading import declaration [24]
3. Highlighting code [28]

E. *Searching*
1. Searching a keyword in webpage [32]
2. File searching [2]
3. Searching by commit hash [1]
4. Searching by commit message [1]

F. *Utilisation of version control*
1. Viewing commit history [17]
2. Reading commit message [11]
3. Resetting working branch to a commit [4]
4. Viewing file changes of a commit [8]

G. *Comparison without the visualisation*
1. Comparing the classes in the same package of between versions (1) [6]
2. Viewing code comparison with diff tool for a class [22]
3. Comparing the same class of different commits (1) [10]

H. *Dependency change identification without the visualisation*
1. Read code to identify changes of dependencies (2)
2. Jot down package names (1)
3. Search package name (2)

I. *Difficulty without the visualisation*
1. Difficult to compare code of different commits on GitHub (1)
2. Difficult to complete task with limited time (1)
3. Troublesome to observe changes in code on GitHub (1)

III. **Benefits of the visualisation**

A. *Ease of use*
1. “Easy to navigate a big project and dive deeper” (1)

2. “History function makes it easy to go back” (1)
3. Easy to compare two versions (3)
4. Easy to see the dependencies (1)
5. Easy to locate a package (1)
6. Easy to view a package (1)

B. *Merits over other tools*
1. Capability of identifying package-level dependency is not provided in existing tools (1)
2. “It makes dependency more obvious/visible compared to other tools” (1)

C. *Supporting change identification*
1. Help check the changes since last commit (1)
2. Help spot changes in role stereotype (3)
3. Help check changes in dependency (2)
4. Capability to show addition and removal of classes (1)
5. Easy to view the changes in code (1)
6. Able to observe the change of number of lines of code (1)

D. *Supporting recognition in interrelation*
1. See how a class is utilised (1)
2. Help understand class relationships (2)
3. Provide overview of system structure (4)

E. *Benefits of role stereotype utilisation*
1. Indication of changing aspects with change of class role stereotype (1)
2. “Colour change in timeline shows the change of role of certain class easily” (4)
3. Help understand the role of classes without code inspection (2)
4. Help expect dependencies (1)

F. *Satisfaction with representations*
1. “Dimmed colour and dotted arrow” (1)
2. Use of graph (1)
3. See dependency by connections of dots (1)
4. “I like how the connections between different components are visualized” (1)
5. Useful metric lines of code (1)
6. Able to observe importance of classes (1)

G. *Quality attributes*
1. “Visualizing the content of the different commits and corresponding code was fast” (1)
2. “The user interface is friendly” (1)
3. “I like the zoom-in and out feature” (1)
4. Provide the code for classes (1)
5. “The tool was very usable and easy to use” (3)
6. “Highlighted newly added code. Very good” (2)

IV. **Improvement on the visualisation**

A. *Difficulty of finding a version*

1. No year indicator on timeline (1)
2. Have to overlay pointer on the shapes on timeline (1)

B. *Problems on representation*
1. Use of pink colour causes confusion (1)
2. "Too many lines prevent me from clicking the package/class" (1)
3. Removed dependency should be in bold colour (1)

C. *Ambiguous components*
1. Vague meaning of icons on toggle button (2)
2. "I missed the role change visualisation part" (2)

D. *Usability problems*
1. Unhandled double click issue (2)
2. "Often jump to other pages/refresh" (1)
3. User guide not shown at the beginning (1)

E. *Incapability of the visualisation*
1. Absence of animation on changes (2)
2. No filtering on versions by year (1)
3. Incapability to remove/filter versions (2)
4. Incapability to open any code repositories (1)
5. No option to change the visibility of dependencies (1)
6. Missing role definition (1)
7. Keep track of the comparing versions in history (1)
8. Incapability to see the selected element with the overall structure (1)

6.2.5 Theoretical Narrative

This section presents the theoretical narrative created using the theoretical constructs, themes and codes listed in Section 6.2.4. The attitude and experience of the research participants are described with the theoretical narrative below, regarding the research concern of how the visualisation tool helped them perform software evolution tasks. The theoretical narrative is separated into four subsections with regard to the four theoretical constructs. The theoretical constructs in capitalised letters and the themes in italics are included in parentheses.

6.2.5.1 Approach with the Visualisation

With the described approaches by the participants in the survey and the observed behaviours from the video recordings, the participants' approaches with the use of the visualisation can be seen (**APPROACH WITH THE VISUALISATION**). The participants had performed various operations of selection in the diagram (***Selection of element in the diagram***), in particular clicking the target package or class, for example, one participant stated that "*I clicked on the package of interest to identify possible changes in dependencies and role*". It was notable that none of the participants attempted to click an untargeted package. Moreover, they also clicked on the background of the diagram to access to the system level. Some of the participants clicked on the background of the diagram accidentally, especially when a dialog was opened. Selection of different elements allowed them to access to various abstraction levels (***Accessing abstraction levels***). One participant

clicked the already selected package multiple times. By clicking a class, package or the background, the view of the class, the package or the system was displayed respectively. During the tasks, the participants accessed to the package level the most frequently, followed by the class and system levels. Most participants had used the mouse to hover over a package or a class node to highlight the selected element and its related elements, as well as dragged a package (***Selection of element in the diagram***). Among all the selecting operations, the most frequently occurring was hovering over a class, followed by clicking a package.

The timeline was the only possible way to change the current viewing version (***Operations on timeline***). Among all the participants' actions, hovering over a version on the timeline was shown as a recurring action, which happened 657 times, with 131 times on average. This was resulted from the hidden version data on the timeline, making the hovering actions unavoidable. The participants used the timeline to select the current version, and also the comparing version (***Selection of comparing version***). Some participants tended to click the versions adjacent to the targeted version on the timeline prior to locating the desired one. Some participants chose to click the targeted element before selecting the specific version, whereas some participants did it the other way around.

There were three ways to perform comparison between two versions in the visualisation (***Ways to compare two versions with the visualisation***). First, the toggle button was used the most by the participants, it was observed that they clicked it 101 times in total. The participants switched between the two modes to select either the current or the comparing version on the timeline, by clicking the toggle button (***Selection of comparing version***). Second, some participants used the compare dialog to select the version for comparison (***Selection of comparing version***). Third, the slider was used by two participants, which had the least number of usage (2 times). The participants clicked a point on the slider or used the previous or next button to select a version (***Selection of comparing version***).

Immediately after selection of the comparing version, the changes of the system, selected package or class were displayed in the diagram, according to the current abstraction level (***Change identification with the visualisation***). The counted number of occurrences shows that the participants mostly viewed the changes at the package level, with 74 times, whereas the numbers for the class and system levels are 18 and 1 respectively. Moreover, albeit uncommon, two participants had viewed the added or removed class list in the compare dialog. Some participants had opened the compare dialog to read the legend of the representations of the changes of class and dependencies, in order to understand the notations of the changes in the diagram, as one participant described his approach: "*I also used the notations and legend to grasp the changes that have been done (types of arrows and different class colours)*".

The research participants identified the dependencies between the elements in the software system using the visualisation (***Dependency identification with the***

visualisation). All participants reported the found changes of dependencies without validating with the code. The dependencies between the classes were examined at the package and class levels. At the package level, the participants identified the package-level dependencies between the selected package and the related packages with arrows in order to find answers to the question 2 of the task. One participant stated, "*Identify dependencies of packages very easily by simply looking at the arrows*" and "*It is much clearer to identify the whole structure and the relationship between packages of the system*". Furthermore, the participants identified the changed dependencies among classes between two versions, in particular the added and removed dependencies. A participant mentioned that "*Instead of having to look at the code to know the dependency I could see if by the connection of the dots, and while comparing version I saw if the dependencies had changed*".

Most participants utilised the role stereotypes when working on the task (***Role stereotype utilisation***). The major usage was to identify the role stereotype of a class by its colour with the use of legend. One of the participants mentioned the use of colour to identify the role stereotype of a class with the purpose of understanding the functionality of the class: "*Identify the brief role of a class by looking its colour, which is helpful in understand the code and hence its functionality*". This also showed that the participant considered both the information of role stereotype and the corresponding code in order to perform the task. Another participant focused on the change of colour for certain class to identify the change of role stereotype: "*using the stereotypes colors, it was easy to spot the change in the role*". Some participants used the timeline component to identify significant change of role stereotype for the selected class, as stated by two participants: "*Look at the change of colour in the timeline to see if there is a big change in the role" and "For the role changes I used the timeline to spot the change in colour indicating the change in stereotype role*". When selecting the comparing version, it was obvious that some participants deliberately pick a version where a change of role stereotype occurred, as shown in the timeline. In contrast to reading the source code of classes in order to answer the question regarding the responsibility of a class, some participants relied on the data of role stereotype to identify the responsibility and its changes over the targeted versions.

The participants used the code feature in the visualisation to view the source code for the investigating classes (***Codes in the visualisation***). The participants expressed "*I have seen the code of the element of interest and used it to make conclusions*" and "*Then use the code function to bring up the code to get a better understanding*". Some highlighted the code while reading the source code of the inspecting class, as well as searching a keyword in the code. One participant mentioned the highlighted differences of the code of the same class between two versions: "*Read highlighted codes to observe the change and new functionality*", to identify the changes on the source code. All the participants had used the code comparison feature to view the changes in the source code of the same class. Nevertheless, only three participants had viewed the source code of a class alone without code comparison.

The participants also used some supporting features provided by the visualisation (***Supporting features***). They used the zooming feature to zoom-in and zoom-out: "*I also zoomed in- and out to better visualize the package of interest and that was helpful*". Some participants had used the history feature to reverse back to the previously viewed elements, instead of selecting the version and locating the package/class again. One participant had downloaded the graph as image once; however, no preceding action relating the image file was observed.

Some unforeseen behaviours were observed (***Unexpected behaviours***). Two participants had performed double clicking on a package or a class, altogether 9 times. Moreover, one of them double clicked a version on the timeline once. Some participants tried to click an edge in the diagram, with 21 times in total. It was observed that the intention of clicking the edges was to click the package behind them. Two participants also attempted to open the source code at the package level, in which no code was available.

6.2.5.2 Approach without the Visualisation

All research participants also worked on a task without the use of the visualisation (**APPROACH WITHOUT THE VISUALISATION**). To start with, they opened the listed code repository on the GitHub website (***Code retrieval***). Two of them chose to clone the code repository using GitHub Desktop, whilst three chose to view it directly on the GitHub website. One participant downloaded the code from the website after attempting to view the code repository on the website.

Five software and tools were used by the participants when working on the task (***Utilisation of tools***). Given that the links to the GitHub code repository of the five commits were provided, all participants inevitably opened the links and used the GitHub website. As mentioned previously, two participants had used GitHub Desktop to retrieve the code repository. One used Sourcetree and Visual Studio Code for viewing the commit history. Two participants used Sourcetree or command line to reset the current branch to a commit. Three participants used Eclipse, Visual Studio Code or Sublime Text to open and view the downloaded files.

Regardless of the tool the participants used, navigation in the code repository was inevitable (***Navigation***). With the purpose of looking for a targeted package folder, the participants had hovered over an untargeted package folder (47 times) and clicked an untargeted package folder (20 times). The participants had clicked a targeted package folder 106 times in all. To locate a class, they hovered over an untargeted class 25 times altogether. In all, they had clicked a class to open the code 129 times. A participant described his process: "*Open the folders in separate tabs, containing the classes and then open the code to see how the changes to dependencies have taken place*". Two participants navigated to another class from the source code. The most frequent behaviour observed was switching tabs, with the sum of 552 times. Two of the participants had performed it over 200 times. Additionally, the participants navigated to the previous page 23 times in total.

All participants had read source code when working on the task without the visualisation, and this was mentioned by all of them when describing their approach (***Code viewing***). One participant stated, "*Read and understand the meaning of the code to understand its role and functionality*". Four participants had focused on the import declaration of the classes in order to answer the questions. The participants also highlighted source code when reading it.

Searching was an action that most participants had performed (***Searching***). They used the search feature of the browser to search a keyword in the webpage. Two participants had performed file searching in Visual Studio Code or Sublime Text. Furthermore, a participant searched for a commit by using the commit hash or the commit message.

All participants utilised the version control components (***Utilisation of version control***). They read the commit messages of the packages/classes/commits. Three participants had viewed the commit history, with 17 times in all. Some participants had reset the working branch to certain commit (4 times), as they put it: "*reset the code to target version*" and "*Checking out the different versions of the commits*". Four participants also viewed the file changes of a commit, as described by one participant: "*Looked mainly at the history and diff function in GitHub and compared codes*".

It was observed that the participants adopted three methods to perform comparison (***Comparison without the visualisation***). The most common method was to view the code comparison with diff tool for a class between two commits (22 times). Another one was manually comparing the code of same class of different commits. Lastly, one participant attempted to compare the class list of a package between the commits, so as to identify the addition and removal of classes occurred in the package.

Three methods to identify the changes of dependency were emerged from the participants' actions and described approaches (***Dependency change identification without the visualisation***). A major one was to read the code: "*Open the folders in separate tabs, containing the classes and then open the code to see how the changes to dependencies have taken place*". Apart from code reading, one participant mentioned the search of package name in the source code of a class: "*To find if there is a dependence to a certain package, use Control-F to search the name of the package*". Another participant adopted similar approach: "*when looking for dependencies, global searching the name of the package and obtain the changes*". One participant also mentioned jotting down the names of package involved in the older version: "*Jot down the packages used for the old version, and see if the packages are removed or new packages are added in the later version*".

Two participants had expressed the difficulty of performing the task without the use of visualisation (***Difficulty without the visualisation***). One participant

brought up the problem of time limitation for performing the evolution task: "*This was a difficult task that I could not complete it within the given time*". The same participant also mentioned the difficulty of comparing the source code of the different commits on GitHub: "*First, I tried to check the code of the different commits on the GitHub website. Then, I realized that it was difficult to make the comparison*" and he later downloaded the source code of different commits and compared the code by using Eclipse. Another participant stated regarding the inconvenience of observing the changes in code on the GitHub website: "*Switch back and forth the websites frequently to observe if there is changes in the code, which is troublesome*". He opened different tabs containing the source code to be compared and switched frequently between the tabs on the browser to spot the changes in code, which he found it troublesome.

6.2.5.3 Benefits of the Visualisation

The participants considered the visualisation was advantageous in three major areas, change identification, recognition in interrelation of the elements in the system and the utilisation of role stereotypes (**BENEFITS OF THE VISUALISATION**). Firstly, the visualisation helped the participants identify the changes in terms of dependency and role stereotype (***Supporting change identification***), as a participant put it, "*The visualization tool provides all the means that can help to check the changes in roles and dependencies*", and "*using the stereotypes colors, it was easy to spot the change in the role*". The same participant pointed out that the visualisation helped check the changes of the system since the last commit version: "*It also helped ... check the changes that have been made since the last commit*". Another participant mentioned the capability of the visualisation to illustrate the addition and removal of classes by the use of dimmed colour and dashed arrow: "*Dimmed colour and dotted arrow show the removal and addition of new classes easily*". One participant believed that the visualisation facilitated the viewing of the changes in codes: "*New codes are highlighted makes it much easier to see what's added and compare the differences*". Although the participant did not apply the lines of code metric during the task, he considered the metric could help observe the change of number of lines of code: "*Metric Lines of Code is useful to observe the importance of the class/the change of number of lines written*".

When it comes to interrelation recognition, most participants agreed that the visualisation provided an overview of the systems structure (***Supporting recognition in interrelation***), for instance, "*It is easier to understand the whole structure*" and "*It is much clearer to identify the whole structure and the relationship between packages of the system*". The participants also expressed that the visualisation helped understand the relationships between classes, as well as showing how a class was utilised by other classes: "*Helped me to understand their relation to other classes and made it easier to see how it was utilized*".

Several benefits of using the visualisation of role stereotypes can be recognised from the participants' answers (***Benefits of role stereotype utilisation***). One benefit was "*Colour change in timeline shows the change of role of certain class easily*", in-

dicating that the timeline component helped the participants identify the change of role stereotype for certain class over the full range of commit versions. Visualising class role stereotypes also helped the participants understand the role and functionality of classes without studying the source code in detail, especially for junior developers: "*Understand the role and its brief functions without careful inspection of the code*". The participant with more programming experience found that the visualisation helped expect the dependencies: "*By knowing the role stereotypes, it becomes easier to expect dependencies*". Another participant mentioned that the changing aspects could be noticed from the change of class role stereotype. The participant described that "*So by understanding that one class has changed the role stereotype, we can have an indication on which aspects are changed for the class in concern. For example, we can see if the class now provides other functionality or if it addresses another concern or responsibility*".

One of the research participants pointed out the usefulness of the visualisation tool compared to other tools (***Merits over other tools***): "*It makes dependency more obvious/visible compared to other tools such as sublime text or other svn applications*" and "*The tool can help in finding dependencies across packages yet such functionality is not provided in existing tools*". The participant valued the visualisation's capability of illustrating the class- and package- level dependencies, with comparison to the tools he knew or used.

The participants described about the easiness of the visualisation tool in performing the tasks (***Ease of use***). One participant mentioned the easiness of locating and viewing a package in the visualisation: "*it is easier to locate and view the specific package*". Moreover, the participants also expressed the easiness of comparing two versions and view the dependencies among the classes in the system: "*By being able to in an easy way see dependencies, versions etc and to be able to compare them*". Navigation within a system was inevitable for both the tasks with and without the visualisation. One participant commented that "*The visualisation is good and it is easy to navigate a big project and dive deeper*" regarding the navigation issue. The participant also mentioned about the history function, which facilitated reversing back to a previously selected view of choice: "*The history function makes it easy to go back*".

The participants showed satisfaction with the representations used in the visualisation in general (***Satisfaction with representations***). One participant mentioned the use of graph to illustrate the elements and their relationships facilitated understanding of the whole structure of the system. Another participant acknowledged the inclusion of the metric lines of code and its usefulness in observing the importance of classes. Two participants expressed their acknowledgment on the notations used to represent the changes of classes and dependencies: "*Dimmed colour and dotted arrow show the removal and addition of new classes easily*" and "*Instead of having to look at the code to know the dependency I could see if by the connection of the dots, and while comparing version I saw if the dependencies had changed*". One participant acknowledged the visualisation of the connections between the elements

in the system: "*I like ... also how the connections between different components are visualized*".

Concerning the quality attributes of the visualisation (***Quality attributes***), one participant expressed that "*Visualizing the content of the different commits and corresponding code was fast*", related to the performance of rendering the graph and retrieving the source code for the selected classes. With regard to the usability of the visualisation, two participants specifically stated that the visualisation was easy to use: "*I think it is intuitive, and has good functionalities*" and "*The tool was very usable and easy to use*". Some participants acknowledged in particular the zooming feature and the inclusion of source code in the visualisation, as well as the user interface: "*The user interface is friendly*" and the highlighting of the newly added code when comparing the source code for the same class of different versions: "*Highlighted newly added code. Very good*".

6.2.5.4 Improvement on the Visualisation

The participants felt that there was still room for improvement of the visualisation tool (**IMPROVEMENT ON THE VISUALISATION**). One thing was the effort of locating a target version (***Difficulty of finding a version***). They had to overlay the mouse pointer on each of the shapes on the timeline in order to locate the specific version they wanted to view or compare. Moreover, one of them suggested the addition of year indicators on the timeline could facilitate the process of locating a version.

In addition to that, they identified some issues on the representations when working on the tasks (***Problems on representation***). Some participants had neglected some elements resulting from the use of pink colour to represent the role Structurer. When the changes between two selected versions were shown, they tended to believe that pink colours were faded red colours, which represented unchanged classes with Information Holder as the role stereotype. One also commented that he regarded the dashed lines which represented removed dependencies as unchanged elements due to the use of a lower opacity. One respondent thought that "*Too many lines prevent me from clicking the package/class*". Many had to zoom in to a package to click it owing to the considerable number of lines between the class nodes.

Some found several components in the visualisation ambiguous (***Ambiguous components***). One participant did not regard the timeline with shapes in different colours indicating the evolution of role stereotypes: "*I missed the role change visualisation part*". The meaning of the icons on the toggle button used to select the current version or the version to be compared was vague to some of the participants.

The visualisation can be improved in terms of usability (***Usability problems***). As one participant put it, the visualisation "*Often jump to other pages/refresh*" since participants tended to perform unwanted clicks on the elements which triggered the re-rendering of the graph. Two participants reported that double clicking the visualisation caused abnormal behaviours. Moreover, the user guide in the visualisation

can be prompted once the visualisation has opened, as suggested by one respondent.

The participants also expected some extra features for the visualisation (***Incapability of the visualisation***). Some of them wanted animation to highlight the changes between versions. The difficulty of finding a version, which was mentioned previously, can be relieved by adding year filters to the versions, as well as the capability of showing only the desired versions/removing unwanted versions, as suggested by two participants: "*Select desired versions to inspect, instead of showing all of them*" and "*add an option to remove some versions from the version history for better organisation*". As most of the participants were unfamiliar with role stereotypes, an introduction to role stereotypes can be included. In addition, one participant wanted to use the visualisation with the data of private code repositories instead of the hard-coded Bitcoin Wallet system used in the user study. The same respondent also suggested adding an option to control the visibility of the arrows representing the dependencies. Another participant expressed that the visualisation should keep track of the two selected versions for comparison in the history list: "*When going back in history it should keep track which versions you were comparing*", so as to reduce the efforts of choosing the comparing version again. One participant wanted to see the overall system structure at all levels and how the current viewing element related to the system structure: "*I think that it would be good to see the overall structure at all times and to visualize the current view in respect to the whole structure*".

7
Discussion

This chapter presents the answers to the research questions stated in Chapter 3.

7.1 Answers to RQ1

This section presents the answers to **RQ1**: *How can data of evolution of class role stereotypes of a software system be visualised?*.

First of all, the six class role stereotypes can be encoded by six different colours. Colour hue is chosen over shape owing to the fact that colour hue is a selective visual variable while shape is not. It is easier to see the distribution of single role stereotype encoded by a selective visual variable rather than the non selective ones (Roth, 2007). Both pink and red colours were used to encode two role stereotypes, nevertheless, the participants regarded the solid pink colour as red in lower opacity, mistakenly neglected the change it represented. Therefore, the adopted colours should be distinct such that the colours are still distinguishable even in reduced opacity.

The Temporal Snapshot strategy identified by Novais et al. (2013) is adopted to visualise the software system at a specific version. To visualise the collaborations between the role stereotypes, directed node-link diagram can be used, in which a node represents a class with the corresponding role stereotype and a directed link represents the dependency between two connecting classes. Moreover, bordered rectangles can be used to illustrate the packages, in which the classes are organised and placed inside. This approach coincides with one of the four types of visualisation techniques, i.e. the graph-based visualisation, identified by Shahin et al. (2014).

Jahnke et al. (2002) emphasise the importance of traversing between abstraction hierarchies smoothly and mention the widely used hierarchical nested graph for showing different levels of abstraction. This approach is consistent with the requirements identified by Jahnke et al. (2002) for navigating between abstraction hierarchies in vertical direction, by using nested sub graphs to show the part-of hierarchies. By clicking the corresponding elements in the graph representing the system, packages and classes, the abstraction level and view can be adjusted. The visualisation can be divided into three levels: system, package and class, so as to visualise the structure and collaborations at different abstraction levels. The system level shows the system structure as a whole and the hierarchical relationships between the packages can be

seen. This level can act as the entry point to access to a specific package or class. For the package level, apart from the classes existing in the selected package, the collaborating classes from other packages can also be included; thus, package-level dependencies can be identified. The class level can display the collaborating classes with different dependency types (incoming, outgoing and both) and within different dependency levels, maximum three levels ideally, otherwise the number of classes shown may be substantial.

The evolution of role stereotype for a class can be visualised by a timeline, which adopts the Temporal Overview strategy identified by Novais et al. (2013). The timeline at the class level is constituted by a sequence of coloured circles representing the class with the corresponding role stereotype over the versions in chronological order, such that the changes of role stereotype can be noticed easily. For the system and package levels, the information of dominant role stereotype (role stereotype having the highest number of occurrences) within the system or the package can be incorporated into the timeline. However, shapes other than circles should be used in order to distinguish from the class level to avoid confusion, for example, a bordered square mimics a package and a square can be used for representing the system.

The approach adopts the Differential Absolute strategy identified by Novais et al. (2013) when displaying the differences between the entities of two comparing versions. Concerning the changes between two versions, four types of class changes and three types of dependency changes are identified. The class changes can be grouped into unchanged, added, removed and role changed. Opacity and the neutral grey colour can be used to form the representations for the changes of class. An unchanged class can be represented by the coloured node with a reduced opacity, whilst the solid colour can be used for an added class. To illustrate a removed class, the colour of the node is changed to grey colour; thus, removed classes can be easily recognised among the coloured nodes. For classes that involve change of role stereotype, adding a coloured border such that the node body and border colours show the two role stereotypes involved. Moreover, the data of the class changes can be displayed as lists in the meanwhile, in which a list item holding a class change can be clicked to help locate the class in the diagram.

To visualise the three kinds of dependency changes, opacity and dashed line can be utilised. Likewise, an unchanged dependency can be represented by the link with a reduced opacity, whereas an added dependency can be indicated by the use of solid colour without reduction in opacity. On the other hand, the removed dependency is represented by using a dashed line instead of solid line. Feedback from the research participants shows that reduced opacity should not be applied to a removed dependency.

By visualising the data of changes of the role stereotype and the collaborating classes, which can be seen by the changes of dependencies, for a specific class in the same view at the class level, a probable indication of transfer of responsibilities from the class to its collaborating classes can be shown.

The data of classes that have changed in terms of role stereotype over the versions can be visualised in the diagram, for example, an animated dashed border around the node can be used to highlight each of the specific classes, and this can be recognised even in a diagram with abundant elements.

There are two suggested ways to illustrate the changes between two selected versions. One is pairwise comparison. Selected a version for comparison, accompanied by the current viewing version, the changes between the two versions are displayed according to the selected abstraction level. Another way is consecutive comparison. When investigating the changes across all the versions, showing the changes between two consecutive versions may be helpful. With the help of a slider and two buttons (previous and next), the changes from the first two versions to the last two versions can be visualised continuously. It can be started from any two consecutive versions by selecting a version on the slider without question, and thereby the changes between the selected version and the preceding version are shown.

Code proximity is a functional requirement for software visualisations identified by Kienle and Müller (2009). Fast access to source code is supported in this visualisation approach, so as to reduce the distance between the representation of a class and the actual entity in source code (Ducasse et al., 2001, as cited in Kienle & Müller, 2009). The source code with syntax highlighted and the highlighted changes of code between two versions can be included to facilitate the investigation of software evolution, in particular the changes of role stereotypes.

Nurwidyantoro et al. (2019) suggest a usage of role stereotypes, which is to find out the patterns of collaboration between role stereotypes. The combination of data of role stereotypes and dependencies can reveal the collaboration patterns of the objects in the software systems. This can be explored by allowing user-defined patterns, suggesting common collaboration patterns, and listing all the identified patterns in the versions. A collaboration pattern is either defined or chosen; accordingly, the diagram displays the resulted collaborations of classes matching the pattern and a line chart can be included to show the changes of the pattern in terms of the number of occurrences over the versions.

According to the classification from Novais et al. (2013), the purpose of this software evolution visualisation approach relates to change comprehension. With the classification of the purposes of software architecture visualisation proposed by Shahin et al. (2014), this approach is in tune with four of the categories, which are “improve the understanding of architecture evolution” (Category 1), “improve the understanding of static characteristics of architecture” (Category 2), “improve search, navigation, and exploration of architecture design” (Category 3), and “support architecture reengineering and reverse engineering” (Category 5). Judging from the architecting activities classified by Li et al. (2012), this visualisation approach supports architecture recovery and architecture understanding. Moreover, this approach provides the evidence based on industrial studies (evidence level 4) in the domains of financial

software and email server software (Shahin et al., 2014).

7.2 Answers to RQ2

This section presents the answers to **RQ2**: *How does the proposed visualisation approach/tool help developers to perform software evolution tasks?*.

7.2.1 SQ 2.1: *Is the proposed visualisation approach/tool helpful?*

The results show the visualisation facilitated navigation within the software system, and helped the participants locate a package or class at ease, whereas the participants were inclined to hover over and open multiple unrelated packages to locate the target class when working on the task without the use of visualisation. The use of graph and the representations for visualising the connections between different components of the software system were acknowledged by the participants.

The proposed visualisation approach enabled the participants to compare two selected versions and identify the changes between them, in terms of the dependency, role stereotype, class, and the number of lines of code. It is worth mentioning the visualisation helped recognise the package-level dependencies and the changes.

In addition to the change identification in comparison of two versions, the visualisation approach also helped recognise the interrelation between the components in the software system. By incorporating the node-link diagram to represent the classes and dependencies, the class relationships can be identified by the directed links between the classes. Bordered rectangles are used to represent the packages in hierarchy. The combination makes the overall system structure and behaviours visible. The data of role stereotypes enriches the visualisation by providing the brief roles of the classes and the collaborations between the classes.

Regarding the usefulness of role stereotypes, the participants stated that the visualisation approach with role stereotypes helped understand the responsibility of the classes without inspecting the source code, especially for those with less experience in programming. The timeline in the visualisation showing the evolution of role stereotypes over the versions helped the participants identify when the changes of role stereotype for certain class occurred, as well as how the role stereotype changes. A participant also suggested the aspects changed could be revealed by understanding the change of role stereotype of a class, for example, the class provides other functionality.

The visualisation tool retrieves the source code for the viewing classes and provides the diff feature to highlight the differences in the source code of the same class between two comparing versions, which saves the navigation time to locate the source code for the two versions, as well as the need to use/install a diff tool with an extra

software. The code feature also serves as a complement to validate the role stereotype of certain class and inspect the functionalities provided by the class.

Judging from the mentioned benefits derived from the feedback and experience of the research participants, it appears that the proposed visualisation approach is helpful in performing software evolution tasks.

7.2.2 SQ 2.2: *What approach do developers use? (with and without the tool)*

The following approaches that developers use to perform software evolution tasks with and without the visualisation tool are summarised from the research participants' responses and actions.

The approach developers adopt with the visualisation is firstly presented. They view the system/package/class at one of the three abstraction levels by clicking the elements in the diagram. However, it is found that some may accidentally click on the background of the diagram, especially when a dialog is opened. A button instead of the background can be used to solve this issue. All the packages can be spotted easily, nevertheless, it might be difficult to click a package when there are substantial number of edges blocking it. Moreover, it is unexpected that developers do not change the layout or the metric when working on the task.

In search for the changes of class-level dependencies (question 1 in the task), developers select a targeted version on the timeline and a targeted package in the diagram in the first step, regardless of the order. They choose the targeted version by selecting a version on the timeline. Afterwards, they choose between the compare dialog and the toggle button with the timeline in order to select a comparing version. They are inclined to use the toggle button and the timeline. However, they tend to hover over the versions on the timeline many times to view the information of a version, and this situation needs to be relieved. After selecting the comparing version, the changes are rendered in the diagram. Developers open the compare dialog to read the legend of the notations of the changes, which may indicate the notations used to represent the changes are not intuitive. They then view the changes of classes and dependencies in the diagram, as well as checking the added and removed class lists.

To look for changes of package-level dependencies (question 2 in the task), developers access to the package level by clicking the targeted package. They repeat the mentioned steps of selecting the current and comparing versions. They pay attention to the packages, in which all the classes have added/removed dependency with the targeted package. Alternatively, the slider is used by some developers; however, it is not useful as expected, probably because only five non-consecutive versions are involved in the task. It may be beneficial when developers are allowed to select only the versions to be investigated, with other unwanted versions removed. Moreover, they use the history feature to go back to a previously viewing package/class. Nevertheless, the comparing versions are not recorded in the history list, and hence

they often need to select the comparing version again. It is learned that all the selected versions should be recorded, especially in visualisations handling comparisons.

When investigating the responsibility of a class (question 3 of the task), developers adopt two different methods. Some of them identify the role stereotype of the class from the colour to understand its responsibility, whereas some read the source code of the targeted class and its collaborating classes to determine its responsibility. They both recognise the collaborating classes by identifying the dependencies with the arrows. None of them validate the dependencies by inspecting the code.

With the intention to identify the significant changes of the responsibility of a class over the versions (question 4 of the task), developers take notice of the timeline, in which the colour changes at certain versions indicate the changes of role stereotype. However, not all developers have noticed the indication, probably owing to the fact that it is not obvious enough or they do not pay attention to the colours in the visualisation. In addition, developers use the code comparison with highlighted changes to identify the changes of source code between two versions, and hence understand the changes of functionality of the class. They also identify the changes of dependencies to recognise the addition and removal of the collaborating classes related to the targeted class.

Without the use of visualisation, the first step for performing a software evolution task is to retrieve the code. Developers do this by downloading the code or directly viewing the code repository on the GitHub website, or cloning the code repository using GitHub Desktop. The approach to identify the changes of package-level dependencies remains unknown as the participants tended to skip the related question in the task, which can probably be attributed to the complexity and the time limitation.

With the purpose of identifying the changes of class-level dependencies across packages between two commits, developers choose to compare the code of the same class from the commits. In order to compare the code of two different versions, various approaches are used. One method that developers use is to download the codes for the two commits and open them with the use of software, for example, Eclipse and Sublime Text, then open each class in the targeted package of the two commits to compare the blocks of import declarations. Alternatively, they click the latest commit message related to each class in the targeted package and search the class name to see the file changes with diff tool on the GitHub website. Another method they use is to firstly reset the working branch to the specific commit using Sourcetree, then open the code using Visual Studio Code and navigate to a class. Secondly, they use the version control component of the software to select a comparing commit to view the code comparison with the equipped diff tool. Regardless of the method, they identify the changes of class-level dependencies by the addition and removal of import declarations. However, by solely looking at the changes of import declarations, classes that involve only static calls may be counted as a dependency.

Developers mainly inspect the source code of the targeted class in order to understand its responsibility. They highlight the code when reading, and sometimes navigate to another class from the code. To recognise the major changes of a class over five versions in terms of responsibility, developers open the class in all the versions and switch between the tabs to identify the changes in the codes. They also compare the class list of the package where the class locates to see if any classes are removed or newly added, which may indicate the transfer of responsibility and new dependencies.

In short, code comparison with diff tool is shown essential for both approaches with and without the use of visualisation. How a commit/version can be found among abundant versions at ease is an issue that needs further investigation. It is interesting to notice that the participants tended to click and hover over untargeted packages and classes when navigating to the target without the visualisation, whereas they can spot the targeted package/class and select right away in the diagram.

7.2.3 SQ 2.3: *How can we improve Rologram?*

Based on the feedback and observations of the research participants, the following improvements can be made to Rologram. Some participants found it hard to access to a package owing to the ample number of lines blocking. The rendered graph is overwhelmed by lines when there is substantial number of dependencies, especially visualising large software systems. Accordingly, an option to change the visibility of the dependencies can be added and all the dependencies can be hidden by default, making the packages and classes easily accessible.

Feedback regarding the representations of changes was collected. One participant was inclined to neglect the removed dependencies resulted from the use of reduced opacity; they were considered as unchanged elements mistakenly. Therefore, solid colour should be used instead to highlight the removed dependencies. Another participant commented on the use of the alike pink and red colours to represent two different role stereotypes. Most participants tended to neglect the solid pink colour nodes which imply added classes in the tasks, since they regard the solid pink colour as the red colour with reduced opacity, which indicates unchanged class that was not the focus of the tasks. Hence the pink colour representing the role stereotype Structurer should be replaced by another colour which is distinct from all other five colours used.

Data of 35 and 37 commit versions of the software systems Bitcoin Wallet and K-9 Mail were applied to the visualisation respectively. The timeline constituted by tens of shapes representing the versions is manageable, although some participants found it difficult to look for a specific version. And yet if applying the visualisation approach in practice, the difficulty of looking up a specific version increases drastically. The total numbers of commits for Bitcoin Wallet and K-9 Mail are 4066 and 9304 respectively, as of the date this report is written. Consequently, the visualisation approach should incorporate a way to enable searching and selecting

only the desired commits, or to filter out/remove unwanted commits for viewing the changes. Consecutive comparisons throughout the history with the slider can also be facilitated without the undesired commits. Moreover, year indicators can be added to the timeline to enhance the experience of searching for a specific version.

Given the need to recognise the changes manually, the visualisation tool can be improved by using animation to demonstrate the changes, in particular the added and removed classes and dependencies; thus, the situation of neglecting some changes may be avoided. With the use of animation, the changes can be shown in sequence. For example, the diagram can firstly show all the classes in both versions, then reduce the opacity of the unchanged classes and dependencies, and highlight the role changing classes. Afterwards, the added elements are highlighted in solid colours. After that, all the removed classes and dependencies are turned to grey colour and dashed lines respectively. In addition, the four types of class changes can be reorganised in specific locations with the help of animation, and hence users can view the changes at ease by groups.

The visualisation only stores the current viewing version and the selected abstraction level and element (system/package/class) in the history list. Observed from the behaviours of the research participants, there is a need to store the two selected versions when visualising the changes in the history list, such that the diagram can be restored to a previous state easily without the need to select the comparing version again.

Some participants probably mixed up the desktop and web environments and performed double clicks within the visualisation, which had caused unexpected behaviours. Double click event was unexpected and not considered during the development of Rologram. The tool can be refined by handling the double click event to avoid undesirable outputs.

Data extraction and manipulation need to be done prior to using the visualisation tool currently, and thus enhancement on the tool is required to support private code repositories. In order to allow users to visualise their own projects, the visualisation tool can be improved by integration with the role stereotype classifying tool (Nurwidyantoro et al., 2019; Ho-Quang et al., n.d.) to extract the data of role stereotypes automatically, and the extraction of dependency data using jdeps.

8

Limitations and Delimitations

The validity and reliability threats identified are discussed in this chapter.

Threats to construct validity. The participants in the user study might have understood the role stereotypes differently, and thus the way of using the visualisation tool might be varied. The threat was alleviated by providing tutorials on role stereotypes prior to the user study.

Threats to internal validity. As the convenience sampling method was adopted for selecting the participants for the user study, they might be biased in favour of the visualisation created by the author. Moreover, most participants were not knowledgeable about the role stereotypes and acquainted with the visualisation before taken part in the user study, which may result in low task performance and affect their perceptions to the visualisation. In order to alleviate the problem, a trial version of the visualisation with a software system not used in the user study was provided to the participants to get acquainted to the visualisation, accompanied by sparing minutes for them to familiarise and ask questions regarding the visualisation at the beginning of the user study.

Threats to external validity. The participants of the user study were recruited through personal network rather than random selection in the population, resulting in impediment and reduction of the generalisability. The user study could be carried out with greater variety of groups of participants through random sampling hereafter. However, the Java, programming and work experience of the participants were varied, as well as their occupations.

It is assumed that the results can be applied to software systems written in Java as the data of two Java open-source software systems were used. Nevertheless, the results can be generalised to other software systems that adopt object-oriented programming since they share the common principles, and the role stereotypes can be applied to them generally.

Threats to conclusion validity. The sample size for the user study was small such that it poses a threat of the theory drawn from the qualitative analysis might not be appropriate. More participants could be recruited in future.

Moreover, the selection of the task questions may have introduced bias favourably for the visualisation tool as questions related to dependencies between classes and

packages were included. However, numerous researches have shown that such information of dependency is necessary for developers (Graduleva & Dahaj, 2017; Russ, 2019). The survey question regarding how role stereotypes help complete the tasks may have introduced positive bias towards the visualisation tool. The threat can be mitigated by formulating the question in a more neutral way in future.

Furthermore, some features in the visualisation tool were not explored during the user study as they were not directly relevant in answering the questions in the tasks, as well as the time limitation of task completion. The range and types of task questions could be more varied to cover all the features in future, so as to fully evaluate the visualisation tool.

Threats to reliability. As the study involved a considerable number of classes in the two software systems, it was not feasible to manually assign the role stereotype to each class. Therefore, the study relied on the data source of role stereotypes from the data generated from a previous study, which poses a threat of reliability of the data. However, the accuracy of the tool that was used to generate the data of role stereotypes was confirmed by comparing the ground truth, which was established by labelling the classes in the systems manually, to the results generated by the tool (Fröding & Ngoc, 2020). Moreover, the role stereotypes for the classes related to the tasks were checked manually in order to alleviate the threat. The data source of dependencies relied on the jdeps tool; however, as jdeps is included in the official Java Development Kit, its reliability should be guaranteed.

9

Conclusion

On the whole, the proposed visualisation approach incorporating the evolution data of role stereotypes appears to be helpful for developers in performing software evolution tasks. The use of node-link diagram seems to help developers understand the system structure and facilitate the navigation to the components in the software system. The capability of pairwise comparison and the timeline may help identify the changes of dependencies and the responsibility of classes. It is also possible to recognise package-level dependencies. However, the tool still needs to improve in certain aspects, including the representations, search of a version, usability and adaptability of private code repositories.

Regarding the approach used with the visualisation, developers can access to different abstraction levels to view the dependencies and role stereotypes of the classes. By switching the current and comparing versions, it is possible to identify the changes in the diagram and the timeline by colours and the representations used, accompanied by the single code view and code comparison.

By observing the participant's actions, it is found that they employed various methods in order to investigate the changes in terms of dependencies and class responsibility without the visualisation tool, regardless of the accuracy. However, it seems that code comparison with highlighted changes is crucial for both approaches with and without the tool.

It is found that the research participants were able to complete all the questions in the task with visualisation, whereas most cannot complete some questions for the task without utilising the visualisation. Moreover, some participants tended to rely on the data of role stereotypes without inspection on the source code. For this reason, the reliability of the data source needs to be ensured.

Some of the features introduced in the visualisation approach were not explored by the research participants due to the irrelevance to the tasks, especially the parts related to collaboration patterns. The evolution of the collaboration patterns of the role stereotypes may be a subject of investigation in future research.

Qualitative analysis was used to analyse the data collected in the user study given the limited sample size; however, it is still interesting to know the extent that visualising the evolution of role stereotypes helps boost the productivity and performance of developers when performing software evolution tasks.

The visualisation approach may be evaluated with the context of complete software evolution tasks hereafter, in which programming is involved. Furthermore, whether the approach or the evolution data of role stereotypes is beneficial in identifying the anomalies in a software system and redesign of system can be investigated.

This study contributes a new way of exploration and comprehension of software systems with the proposed visualisation approach, as well as looking into the usefulness of changes of role stereotypes in understanding software evolution. This empirical piece of software evolution visualisation with the data of role stereotypes is added to the field of software visualisation, providing a new perspective of studying object-oriented software systems. By visualising the evolution data of role stereotypes, the proposed approach indicating when the objects in software systems change in terms of responsibility can provide researchers insights into studying various quality aspects of software systems for improvement of system structure in future.

References

Auerbach, C. F., & Silverstein, L. B. (2003). *Qualitative Data: an Introduction to Coding and Analysis.* New York University Press.

Bani-Salameh, H., Ahmad, A., & Bani-Salameh, D. (2016, August 21-25). Software Evolution Visualization Tools Functional Requirements: a Comprehensive Understanding. ICSEA 2016, *The Eleventh International Conference on Software Engineering Advances*, Rome, Italy (pp. 196-200). International Academy, Research, and Industry Association.

Diehl, S. (2007). *Software Visualization: Visualizing the Structure, Behaviour, and Evolution of Software.* Springer.

Eick, S. G., Graves, T. L., Karr, A. F., Mockus, A., & Schuster, P. (2002). Visualizing Software Changes. *IEEE Transactions on Software Engineering, 28*(4), 396-412.

Fröding, F., & Ngoc, D. N. (2020). *The Evolution of Role-Stereotypes and Related Design (Anti)Patterns* (Bachelor's thesis, University of Gothenburg). Retrieved from https://gupea.ub.gu.se/bitstream/2077/67098/1/gupea_2077_67098_1.pdf

Graduleva, A., & Dahaj, M. A. (2017). *Visualization of Software Architecture Based on Stakeholders' Requirements: Empirical Investigation Based on 4 Industrial Cases* (Bachelor's thesis, University of Gothenburg). Retrieved from https://gupea.ub.gu.se/bitstream/2077/52655/1/gupea_2077_52655_1.pdf

Hevner, A. R., March, S. T., Park, J., & Ram, S. (2004). Design Science in Information Systems Research. *Management Information Systems Quarterly, 28*(1), 75-105.

Hevner, A. R. (2007). A Three Cycle View of Design Science Research. *Scandinavian Journal of Information Systems, 19*(2), 87-92.

Holt, R., & Pak, J. Y. (1996, November 10-11). GASE: Visualizing Software Evolution-in- the-Large. *Proceedings of WCRE'96: 3rd Working Conference on Reverse Engineering*, Monterey, CA, USA (pp. 163-167). doi:

10.1109/WCRE.1996.558900.

Ho-Quang, T., Nurwidyantoro, A., Chaudron, M. R. V., Froding, F., & Ngoc, D. N. (n.d.). *Using Machine Learning for Automated Classification of Class Responsibility Stereotypes in Software Design* [Under submission].

Ho-Quang, T., Bergel, A., Nurwidyantoro, A., Jolak, R., & Chaudron, M. R. V. (2020, September 28-29). Interactive Role Stereotype-Based Visualization To Comprehend Software Architecture. *8th IEEE Working Conference on Software Visualization (VISSOFT)*, Adelaide, Australia (pp. 122-132). IEEE Computer Society.

Jahnke, J. K., Müller, H. A., Walenstein, A., Mansurov, N., & Wong, K. (2002, June 27-29). Fused Data-Centric Visualizations for Software Evolution Environments. *In Proceedings of the 10th International Workshop on Program Comprehension (IWPC'02)*, Paris, France (pp. 187-196). IEEE Computer Society, USA. doi: 10.1109/WPC.2002.1021340.

Kienle, H. M., & Müller, H. A. (2007). Requirements of Software Visualisation Tools: A Literature Survey. *4th IEEE International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT)*, Banff, AB, Canada (pp. 2-9). doi: 10.1109/ VISSOF.2007.4290693.

Lanza, M., & Ducasse, S. (2002). Understanding Software Evolution Using a Combination of Software Visualization and Software Metrics. *LMO'02: Langages et Modèles à Objet, 8*(1-2), 135-149. doi: 10.3166/objet.8.1-2.135-149.

Li, Z., Liang, P., & Avgeriou, P. (2012). Application of Knowledge-Based Approaches in Software Architecture: A Systematic Mapping Study. *Information and Software Technology, 55*(5), 777-794. doi: 10.1016/j.infsof.2012.11.005.

Nam, D., Lee, Y. K., & Medvidovic, N. (2018, May 27-June 3). EVA: A Tool for Visualizing Software Architecture Evolution. *2018 ACM/IEEE 40th International Conference on Software Engineering: Companion (ICSE'18 Companion)*, Gothenburg, Sweden (pp. 53-56). doi: 10.1145/3183440.3183490.

Novais, R. L., Torres, A., Mendes, T. S., Mendonça, M., & Zazworka, N. (2013). Software Evolution Visualization: A Systematic Mapping Study. *Information and Software Technology, 55*(11), 1860–1883. doi: 10.1016/j.infsof.2013.05.008.

Nurwidyantoro, A., Ho-Quang, T., & Chaudron, M. R. V. (2019, April 15-17). Automated Classification of Class Role-Stereotypes via Machine Learning. *In Proceedings of the Evaluation and Assessment on Software Engineering (EASE'19)*, Copenhagen, Denmark (pp. 79-88). Association for Computing Machinery. doi: 10.1145/3319008.3319016.

Peffers, K., Tuunanen, T., Gengler, C. E., Rossi, M., Hui, W., Virtanen, V., & Bragge, J. (2006, February 24-25). The Design Science Research Process: A Model for Producing and Presenting Information Systems Research. *Proceedings of 1st International Conference on Design Science Research in Information Systems and Technology (DESRIST 2006)*, Claremont, CA, USA (pp. 83-106).

Peffers, K., Tuunanen, T., Rothenberger, M. A., & Chatterjee, S. (2007). A Design Science Research Methodology for Information Systems Research. *Journal of Management Information Systems, 24*(3(Winter 2007/2008)), 45-77. doi:10.2753/MIS0742-1222240302.

Roth, R. E. (2007). *Visual Variables.* doi: 10.1002/9781118786352.wbieg0761.

Russ, C. (2019). Surviving Software Dependencies. *Communications of the ACM, 62*(9), 36–43. doi: 10.1145/3347446.

Rysselberghe, F. V., & Demeyer, S. (2004, September 11-14). Studying Software Evolution Information by Visualizing the Change History. *Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM'04)*, Chicago, IL, USA (pp. 328-337). doi: 10.1109/ICSM.2004.1357818.

Saldaña, J. (2013). *The Coding Manual for Qualitative Researchers.* Sage Publications Ltd.

Shahin, M., Liang, P., & Babar, M. A. (2014). A Systematic Review of Software Architecture Visualization Techniques. *Journal of Systems and Software, 94*, 161-185. doi: 10.1016/j.jss.2014.03.071.

Wirfs-Brock, R. J. (2006). Characterizing Classes. *IEEE Software, 23*(2(March/April 2006)), 9-11. doi: 10.1109/MS.2006.43.

Wirfs-Brock, R., & McKean, A. (2002). *Object Design: Roles, Responsibilities, and Collaborations.* Addison Wesley.

Wirfs-Brock, R., Wilkerson, B., & Wiener, L. (1990). *Designing Object-Oriented Software.* Prentice Hall.

A

Appendix

A.1 Questionnaire

1. **Background**
   - Age
   - Gender
   - Title of current/last IT job position
   - Java experience (year)
   - Programming experience (year)
   - Work experience in IT industry (year)
   - Experiences in the following areas:
     - ☐ Software development
     - ☐ Software maintenance
     - ☐ Software testing
     - ☐ Refactoring
     - ☐ Debugging
     - ☐ Android development
     - ☐ iOS development
   - Experience to the Bitcoin Wallet system:
     - ☐ I have never used the system or the app Bitcoin Wallet
     - ☐ I have used the app Bitcoin Wallet
     - ☐ I have contributed to the system source code
     - ☐ I have studied the source code
     - ☐ Other (Please specify)

2. **Tasks**

   Task 1

   Answer the following questions related to the package `wallet.service`.

   Complete the tasks related to the listed commits.
   - fe59b12 on 2012-04-02
   - 77fc50b on 2012-07-02
   - 5276722 on 2014-10-03
   - 5a079e6 on 2018-04-01
   - a0b116d on 2018-07-06

i. Spot two dependencies that are removed or added between the classes in the package `wallet.service` and other package between the commits 5276722 on 2014-10-03 and 5a079e6 on 2018-04-01.
For example, the dependency from class A1 in package A to class B1 in package B is added or removed.

ii. Briefly describe the changes of package-level dependency for the package `wallet.service` and mention when did the changes happen (commit id/date).
For example, on 2021-05-01, the dependency from package A to B/from package B to A is added or removed.

iii. What is the functionality/responsibility of the class `BlockchainService` in package `wallet.service` in the commit fe59b12 on 2012-04-02?
The responsibility of a class implies the role the class plays in the system, and in collaboration with other classes.

iv. Describe the major changes on the class `BlockchainService` in package `wallet.service` in terms of the functionality/responsibility over the five commit versions.

Task 2
Answer the following questions related to the package `wallet.ui.send`.

Complete the tasks related to the listed commits.
- f739bc8 on 2014-07-02
- 08a5850 on 2016-10-05
- 5730c1c on 2018-01-11
- 5a079e6 on 2018-04-01
- ce1a108 on 2019-01-18

i. Spot two dependencies that are removed or added between the classes in the package `wallet.ui.send` and other package between the commits 5a079e6 on 2018-04-01 and ce1a108 on 2019-01-18
For example, the dependency from class A1 in package A to class B1 in package B is added or removed.

ii. Briefly describe the changes of package-level dependency for the package `wallet.ui.send` and mention when did the changes happen (commit id/date).
For example, on 2021-05-01, the dependency from package A to B/from package B to A is added or removed.

iii. What is the functionality/responsibility of the class `SendCoinsActivity` in package `wallet.ui.send` in the commit f739bc8 on 2014-07-02?
The responsibility of a class implies the role the class plays in the system, and in collaboration with

other classes.

iv. Describe the major changes on the class `SendCoinsActivity` in package `wallet.ui.send` in terms of the functionality/responsibility over the five commit versions.

3. **Equipment**
   - Operating system:
     ◯ Windows
     ◯ MacOS
     ◯ Other (Please specify)
   - Web browser:
     ◯ Safari
     ◯ Google Chrome
     ◯ Firefox
     ◯ Opera
     ◯ Internet Explorer
     ◯ Other (Please specify)
   - Software and tools used:
     ☐ IntelliJ IDEA
     ☐ Eclipse
     ☐ NetBeans
     ☐ GitHub website
     ☐ Other (Please specify)

4. **Usefulness of role stereotypes**
   (a) *The visualisation of role stereotypes helps understand the responsibility of the classes.*
   ◯ Strongly disagree ◯ Disagree ◯ Neutral ◯ Agree ◯ Strongly agree
   (b) Please elaborate on how role stereotypes help complete the tasks.

5. **Usefulness of the visualisation**
   (a) *The visualisation helps understand the system structure.*
   ◯ Strongly disagree ◯ Disagree ◯ Neutral ◯ Agree ◯ Strongly agree
   (b) *The visualisation facilitates better understanding of the system structure than the conventional approach I used.*
   ◯ Strongly disagree ◯ Disagree ◯ Neutral ◯ Agree ◯ Strongly agree
   (c) *I prefer to use the visualisation for understanding the evolution of a software system than the conventional approach I used.*
   ◯ Strongly disagree ◯ Disagree ◯ Neutral ◯ Agree ◯ Strongly agree
   (d) *The visualisation helps in completing the tasks.*
   ◯ Strongly disagree ◯ Disagree ◯ Neutral ◯ Agree ◯ Strongly agree
   (e) Please elaborate on how the visualisation helps complete the tasks.

6. **Approach of completing the tasks**
   (a) Describe the approach with the visualisation.

(b) Describe the approach without the visualisation.

7. **Feedback on the visualisation**
   (a) Please comment on the visualisation regarding usability and other aspects you could think of.
   (b) Please provide some suggestions to improve the visualisation. Think of any missing features you would like to have.