Universiteit Leiden

Computer Science

Adding Value to
Software Development and Maintenance
by Software Architecture Evolution Visualization

Name: Shenshen Fan
Date: 30/08/2016

1st supervisor: Dr. Werner Heijstek
2nd supervisor: Dr. A.J. (Arno) Knobbe

MASTER'S THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Adding Value to Software Development and Maintenance by Software Architecture Evolution Visualization

Shenshen Fan

First company supervisor: Dr. Haiyun Xu

Second company supervisor: Drs. Xander Schrijen

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University

September 2016

Acknowledgements

There are a number of people I would like to thank for supporting and helping me with the thesis.

Foremost, I would like to express my gratitude to my university supervisors, Dr. W. Heijstek and Dr. A.J. (Arno) Knobbe for all the guidance and feedback on my thesis. Furthermore, I would like to praise my company supervisors Haiyun Xu and Xander Schrijen for providing me with this research opportunity and valuable advice and tutoring during the thesis project. They were always supporting my work, providing extremely useful recommendations in every single step of the research. Their guidance was crucial for completing this thesis. I would also like to thank the people at the Software Improvement Group (SIG) for the kind support during the internship. Lastly, I would like to thank all of the people that contributed to the result of this study by agreeing to be interviewed and participating the validation.

Abstract

Software needs to evolve overtime to cope with continuously changing requirements. “Software quickly becomes very complex when its size increases, which results in a boost in the time and effort needed to understand the system, maintain its components, extend its functionality, debug it and write tests for it”[10]. Therefore, software evolution causes burdens to both development and maintenance process.

Software component organization is referred to as software architecture. A good architecture can help to satisfy the requirements, simplify the understanding of the software and provide a blueprint for development[18]. Thus, a good architecture can help to reduce the burden caused by software evolution. However, software architecture is abstract and intangible. Visualization is an effective way to display and analyze architecture by providing visual representations of software elements. Therefore, this research seeks to develop a new visualization technique for *implemented* software architecture[8] evolution that will be beneficial to both software development and maintenance.

In order to achieve the goal, we answered the “why”, “what” and “how” aspects of visualization design: Why we need the visualization? What could be visualized? How to visualize the architecture? From a literature review and three stakeholder interviews, a list of architectural features and different stakeholders’ needs were identified. The stakeholders consist of developers, software consultants and managers of development. Based on the results, we matched stakeholder requirements with existing visualization techniques and selected features to visualize, which are component and dependency evolution. After that, a visualization prototype was designed and developed. We then validated the prototype through three case studies in which we discussed the prototype’s usefulness and applicability.

In conclusion, this research has three results. First, 12 architectural features were identified as benefiting visualization. Second, requirements from developers, consultants and managers were collected and matched with some existing visualization techniques. Third, this research contributes a new visualization technique for software architecture evolution.

In future research, the results can be improved by collecting the requirements from a larger group of stakeholders which consists of more diversified roles such as architects and

testers. Additionally, the prototype can be enhanced with the function to compare designed architecture and implemented architecture.

Table of contents

List of figures

List of tables

Chapter 1

Introduction

"Advances in science and commerce have often been characterized by inventions that allow people to see old things in new ways" [9]. The development of computer hardware breeds the field of information visualization which enables people to think in an innovative way.

Using charts to help explain and think has a long history. For example, a long time ago, people started to use maps to depict the relationships between the elements of some space and find the right way from one place to another. People used to build a connection between what we see and what we think with the help of visual metaphors. In other words, Visualization is used as an external aid to enhance our cognitive abilities.

With the development of computer hardware, a new medium for visualizations is provided. Computer-based visualization enables real time, interactive and explorative visualizations. To be more specific, computers can help to get real time data and assemble big amounts of data into one graph within a short time. In addition to help understand the knowns, they also allow users to interact with the graph to explore the unknowns. Computer-based visualization has been used in a large number of areas including science, business and education. Among them, software visualization has drawn a lot of attentions. As stated by the fourth IEEE Working Conference on Software Visualization, software visualization is a broad research area encompassing concepts, methods, tools, and techniques that assist in a range of software engineering and software development activities. Covered aspects include the development and evaluation of approaches for visually analyzing software and software systems, including their structure, execution behavior, and evolution (VISSOFT 2016).

1.1 Problem Statement

Software has been an inseparable part of our daily life as we use the services of software everyday even without realizing it. For example, in the supermarket, you make the payment through your bank card. The payment requires the cooperation among multiple systems of the supermarket, the bank and other parties. In fact, there are numerous software systems running all the time to support our life. Thus the smooth operation of software has a significant impact.

All the software is built based on its architecture. Software architecture is defined as "the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them" [2]. Software architecture provides a high level abstraction of software's structure and influences nearly all the quality attributes. Without a proper architecture, the development and maintenance will be difficult and even the smooth operation of software can't be ensured. For instance, system A is composed of several components. In system A, all the other components depend on one component X which controls the business logic. When the "feature creep" increase, component X quickly becomes giant and may cause big problem for its operation. In this case, A's architecture has poor scalability which makes A easily go wrong when its functionality grows.

Getting software right is not an easy task because software is always intangible and what's more, it needs to evolve over time to cope with continuously changing requirements. As described by the laws of software evolution [34], evolving software will experience continuous changing, increasing complexity and quality decay. "Software quickly becomes very complex when its size increases, which results in a boost in the time and effort needed to understand the system, maintain its components, extend its functionality, debug it and write tests for it" [10]. Therefore, the evolution of software results in big burden on software development and software maintenance.

Software visualization aims at improving software understanding by providing visual representations of software related data. It makes software more tangible and therefore more comprehensible and analyzable. Software comprehension both during and after software development is an crucial component of software process [49]. For example, as stated by Holten [26], up to 60% of a software maintenance task is spent on understanding the software. Thus, software visualization can relieve the burden of software development and maintenance. Furthermore, software architecture is a major factor in satisfying the requirements, simplifying the understanding of the software and providing a blueprint for development [18]. Software architecture visualization provides a global overview of the system, which contributes a lot to decide which components need further investigation and to

focus on a specific part of the software without losing the big picture [11]. Therefore, this research focuses on applying visualization techniques to software architecture evolution.

1.2 Research Questions

Prior research has already invented many visualization techniques covering software structure, behavior and evolution (for overviews see [11] and [14]). However, prior research mainly focuses on what data can be visualized and how to visualize them. Questions as to what can be answered by the visualization, why we need the visualization, how effective is the visualization still need to be further analyzed. In order to answer these questions, different stakeholders should be considered since not only architects but also developers, testers, project managers, and even customers can benefit from software visualization [10].

Furthermore, prior research mainly looks at designed architecture which is designed and drafted before implementation as the development blueprint (for example [21] [29]). However, in practice, implemented architecture doesn't always strictly follow designed architecture due to the changing context or requirements. In addition, it's not easy to visualize and analyze designed architecture in some cases. For example, the documentation might be missing or out of date. Therefore, evaluating software architecture without taking the implemented architecture into account is incomplete [8].

The research questions derived and the objectives are listed below.

Question 1: *What are the main features of the implemented software architecture that matter for stakeholders involved in software development and maintenance?*
Objective: To identify the main architecture related features that are important to stakeholders involved in software development and maintenance.

Question 2: *Among existing visualization techniques, what are the best candidates for the identified features?*
Objective: To identify the best visualization techniques to fit stakeholders' needs.

Question 3: *What is involved with implementing a visualization technique, which fits multiple stakeholders' needs of architecture evolution?*
Objective: To build a prototype that can assist multiple stakeholders.

Question 4: *What can be learned from the visualization prototype?*

Objective: To validate the visualization technique, identify what information can be delivered through visualizing the features.

1.3 Research Design and Methodology

In order to solve the raised questions, we proposed a research framework as shown in figure 1.1. When designing a visualization, three key questions should be taken into account: Why do we need the visualization? What data should be visualized? How to visualize the data? To answer the "Why" question, key stakeholders should be identified. After that, we will implement requirements analysis through stakeholder interview. Based on stakeholder's requirements, we will then determine the data to be visualized. Once the goal is determined and the data is ready, we will study the existing visualization techniques from published papers to learn what can be improved and what can be reused. Based on the findings, we will then design an adapted visualization technique which fits multiple stakeholder's needs and develop a prototype. The last step is to validate the usefulness and applicability of the prototype through several cases studies.

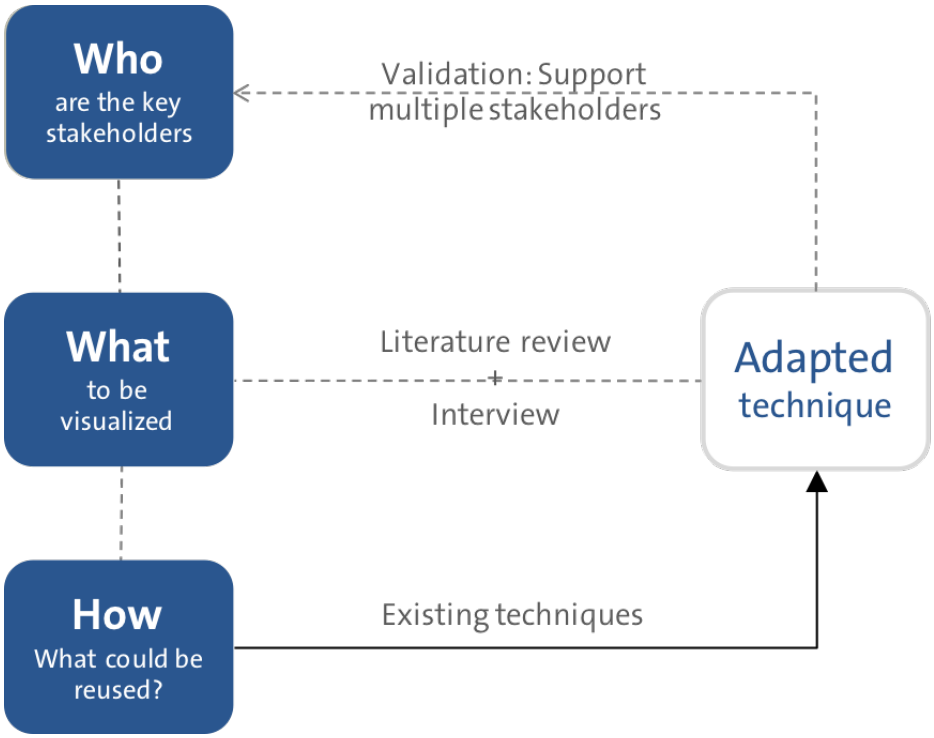


Fig. 1.1 Research framework

Question 1 and 2 are intended to study the "Who" and "What" perspective. In order to answer these two questions, literature review will be conducted to find potential features

from previous research and at the same time, stakeholders' opinion will be gathered through interviews. Furthermore, a group of existing software visualization techniques will be studied to match with different requirements.

Question 3 looks at the "How" perspective in the research framework. In order to answer it, a visualization prototype will be built. Firstly, the related data determined by the results of the first question will be gathered with the help of SIG's (Software Improvement Group)[1] internal tools. After that, we will visualize the gathered data to fit multiple stakeholders' needs.

Question 4 concerns with the validation of the visualization prototype, which is going to be answered by case studies. Several industrial systems will be selected and the prototype will be applied on the chosen systems. The results of visualization will be presented to the stakeholders to identify what information can be derived from the visualization and how this information can help in the software engineering process.

1.4 Thesis Structure

This thesis is structured as follows. Chapter 2 describes the background and context of this research, concerning with software architecture, software evolution and software visualization. Chapter 3 presents existing visualization techniques with the discussion of their pros and cons, which provides a foundation for following chapters. Chapter 4 introduces the design of the visualization prototype covering requirement analysis and functionality design. The visualization design is described in chapter 5. Based on the design, chapter 6 presents the implementation details and final results of the visualization prototype. After that, several case studies are described to validate the prototype introduced in chapter 7. Finally, conclusions are drawn and future work is proposed in chapter 8.

[1] https://www.sig.eu/en

Chapter 2

Background

In this chapter, the background and context of this research is described. It covers three parts: software architecture, software evolution and software visualization.

2.1 Software Architecture

As the size and complexity of software system increases, the design concerns moved from algorithm and data structures to overall design principles which is typically referred to as software architecture. Software architecture constraints almost every aspect of software systems including "gross organization and global control structure; protocols for communication, synchronization, and data access; assignment of functionality to design elements; physical distribution; composition of design elements; scaling and performance; and selection among design alternatives", etc [19].

There have been numerous definitions of software architecture since it was first introduced in the 1960s. For example, according to Kogut and Clements [31], "software architecture is the organizational structure of a software system including components, connections, constraints, and rationale". Bass et al. [2] defined the software architecture of a program or computing system as "the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them".

Although the definition might not always be the same, the importance of software architecture is widely recognized, as stated by David Garlan [18], software architecture works as the bridge between requirements and implementation. As the bridge, software architecture needs to balance stakeholders' requirements including both technical and operational requirements, ensure all the quality attributes and support decision making. The role of bridge can be further specified as the following six functions:

- **Understanding:** software architecture provides high level abstraction of a system, which can help people to understand the software.
- **Reuse:** architectural determines the structure of components which supports both reuse of large components and component integration framework. For example, for a system using pipe-and-filter architecture style [19] as shown in Figure 2.1, the pump is the data source; the filters transform or filter the data they received via the pipes; the pipes are connectors that pass data from one filter to the next; the sink is the data target which stores the output. The filters can be kept small and built with a small set of assumptions about the environment that they operate in. In this way, the filters have big potential to be reused.
- **Construction:** architecture works as a blueprint for development.
- **Evolution:** architecture separates functionality concerns of components and clearly defines the interaction between components. Hence, when a change happens, the related components should be changed as well. In this way, architecture exposes the way software evolves.
- **Analysis:** architecture analysis brings lots of insights for software analysis. For example, quality attributes analysis, dependencies analysis, components behavior analysis etc.
- **Management:** Software architecture facilitates management by improve the understanding of requirements, implementation strategies, and potential risks [18][5], in this way, it can also provide a lot of insights for future decision making.

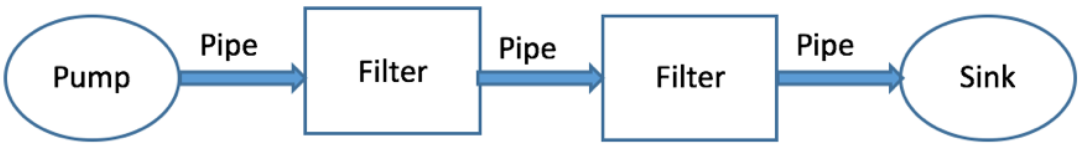


Fig. 2.1 Pipe-and-filter architecture

Software architecture has been a well-established field in recent years and continued to be an important topic in academia. There are several trends within software architecture research as mentioned by Bosch et al. [6].

- The role of architecture is more explicitly recognized in all phases of software development. Initially, architecture was only related to design phase. Now it also involves in development, product derivation in software product line and evolution phase.
- Architecture design is more closely related to stakeholders' requirements.
- Quantitative assessment of architecture is increasingly used. In terms of architecture assessment, quality attributes are heavily used. Traditionally, the assessment of quality attributes was qualitative, but now, researchers tend to quantify them.

Since this research highlights software architecture and its evolution, it's believed that the research will contribute to multiple aspects of software development and maintenance as we mentioned here. Furthermore, this research also starts with stakeholder's requirements and tends to provide quantitative assessment for software architecture.

2.1.1 Software Architecture Concepts Used in This Thesis

In this section, the software architecture related concepts that used in this thesis are described. Bouwers [8] defined module and component as follows:

*"A **module** is a logical block of source code that implements some sort of functionality. In object oriented programming languages this normally is a class, whereas in imperative languages the usual module is normally the file."*

*"A **component** is a coherent chunk of functionality within a system such as a package in Java or a namespace in C#."*

2.2 Software Evolution

Since the requirements towards software are always changing, software must be adapted and enhanced continually to adequately fulfill its roles and satisfy stakeholders' needs. In other words, software must evolve all the time to cope with the changing requirements. As defined by Lehman and Belady [33], "Software Evolution is the process of continual fixing, adaptation, and enhancements to maintain stakeholder satisfaction".

Software evolution is inevitable and it causes a lot of side effects such as software quality decreasing, software aging [39]. As stated by the laws of software evolution [34], software evolution will bring the following negative effects:

- **Continuing change:** Without continual adaption, software systems become less satisfactory and problematic. Continuing change is expensive both in terms of time and budget. In addition, it also makes the system unstable and therefore unpredictable.
- **Complexity:** Along with the evolution, the complexity increases unless proper maintenance is implemented.
- **Continuing growth:** The functional content of the system must be continually increased to cope with changing requirements over its lifetime. Thus the size of software will continually increase.
- **Declining quality:** The quality of systems will decline unless they are properly maintained.

Therefore, proper software adaption and evolution is inevitable and challenging, one should consider both technical issues such as preserving system quality attributes, supporting model evolution and operational issues such as development process management, little management awareness [36]. To make sure the smooth operation of software evolution, techniques and tools which support active maintenance and solve the problem caused by evolution are urgently needed. Visualize the evolution history is one of the effective techniques which attracted a lot of attention in both industry and academia [44].

Software evolution analysis can be done in various ways, for example, software architecture evolution. As stated before, architecture provides high level abstraction of the software systems and balances different stakeholders' requirements, thus the evolution of architecture provides a holistic overview of software evolution with the tradeoff between quality and business goals. Through software architecture evolution analysis, issues about architecture erosion and evolution pattern can be addressed.

2.3 Software Visualization

"Information visualization refers to the use of computer-supported, interactive, visual representations of abstract data to amplify cognition" [9]. The main purposes of information visualization could be communicating a story or exploring unknowns from existing data by mapping raw data into visual representations. The ultimate goal is to improve and accelerate the users' perception by taking advantage of Human Visual System (HVS). The advantages are summarized below [26][46]:

- HVS has high bandwidth, we acquire more information through vision than through other senses. HVS enables people to process visual information in a preattentive way

so that visualization not only helps people to process big amount of data rapidly and also allows people to focus on the information we want to deliver.

- HVS has excellent pattern recognition capability and allows for the easy perception of unexpected features. In other words, visualization makes the pattern and anomalies within the data apparent. In this way, visualization can improve people's capability to identify visual patterns and anomalies.

In conclusion, visualization improves people's perception and understanding of the data especially big data. With the developing of computer graphics techniques, visualization has been widely used to presenting data.

Evolving software quickly becomes large and complex, as a result, a huge amount of information at levels of all granularity is coming out. Therefore, using visualization to understand and analyze software is value adding. The field of software visualization concerns with visualizing the artifacts of software and its development process [14] which includes structure (and additional metrics), behavior and evolution of software as shown in Figure 2.2.

Structure and Metrics: Structure refers to the static aspect of software that can be inferred without running the program. It focuses on the source code and data structures, the static relations between elements. Software metrics usually will be displayed as additional attributes on structure visualization to indicate the quality of software. For example, LINES_OF_CODE measures the size of software elements and is often displayed besides the elements. Software architecture visualization is a part of structure visualization.

Behavior: Behavior visualization is to visualize the dynamic aspect of software which provides information of a particular run on software. For each run, the source code and runtime data such as parameter values are stored and visualized. The results could be a high level abstraction of algorithms behavior or communication between objects. Animating algorithm is one of example, it refers to the process of abstracting data, operations and then animate the behavior of them so that the algorithm can be better explained and perceived by users.

Evolution: Evolution refers to the code evolution of the software and aims at facilitating the software maintenance by visualizing how a system came into being. It focuses on how the code has been changed such as new code, removed code and changed code. During evolution, people are not only interested in the modules that change frequently but also want to know about the stable modules which are rarely changed. In general, there are two widely used ways to visualize evolution. One is to add a time dimension to static

visualization using animation and interaction technique to switch from one state to another. In this way, the transition between two states is presented and it increases the coherence of consecutive states. The other one is to display all the changes in a single graph [11]. It provides an overview of all the changes and reduces the cognitive load since users don't need to remember previous states.

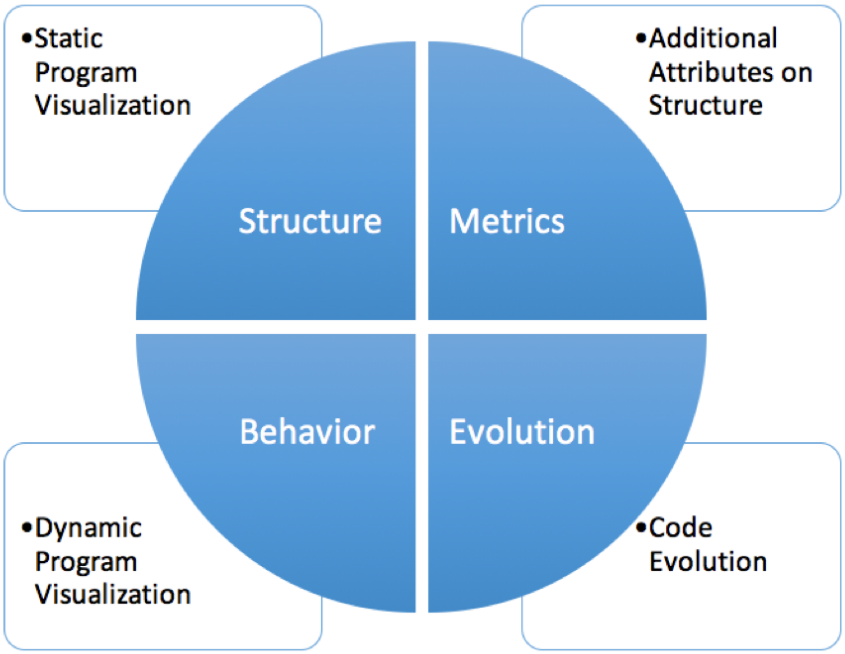


Fig. 2.2 The branches of software visualization

Chapter 3

Literature Review

To solve the "What" and "How" question, a set of relevant papers and existing visualization techniques were studied. We put emphasis on several aspects: the architecture features visualized, the way to visualize the data and the strengths or limitations of the techniques.

In Section 2.3, we introduced two approaches to visualize software evolution [11]:

- Add a time dimension to the static visualization which refers to structure and metrics visualization.
- Display the entire evolution in a single graph

Therefore, in this research, techniques for the visualization of software structure, metrics and evolution are studied, and visualizations of behavior are out of scope.

3.1 Static Architecture Visualization

Generally, object-oriented software is constructed hierarchically with components containing classes, and recursively with classes containing units. Hence, software architecture visualization techniques normally consist of visualizations of software's hierarchical organization and the relationships between software elements [11]. Software metrics will be added to both of these two visualizations as external attributes. Thus in this section, we introduce visualization techniques from the first two perspectives.

3.1.1 Organization Visualization

Treemap, first introduced by Johnson and Schneiderman [27], is an optimized way of using the tree metaphor to display the entire software hierarchy with the goal of maximizing the

space usage. As shown in Figure 3.1 (a), Treemap recursively slices boxes into smaller boxes to display the whole hierarchy. Each of the boxes represents an element, the containment relationship among the boxes indicate the hierarchy of the software. As a result, all the elements of hierarchy are displayed using nested boxes. In terms of software architecture, Treemap visualizes units as elementary boxes, classes as composed boxes and components as the combination of composed boxes. In addition, the attributes of the boxes such as width, length and color are used to encode the elements' attributes which could be metrics. For instance, the size of each boxes can represent the size of the elements which can be measured by LINES_OF_CODE. This provides a symbolic idea of how to combine structure and metrics.

Generally speaking, "Treemaps provide an extremely compact layout, however, they are limited by mainly showing the leaves of the software structure" [30]. Since the hierarchical structure are not explicitly displayed, it causes difficulties to discern the hierarchy level. Another drawback is that the elements are hard to differentiate since all of them have the same shape. In order to eliminate the limitations, researchers have come up with several

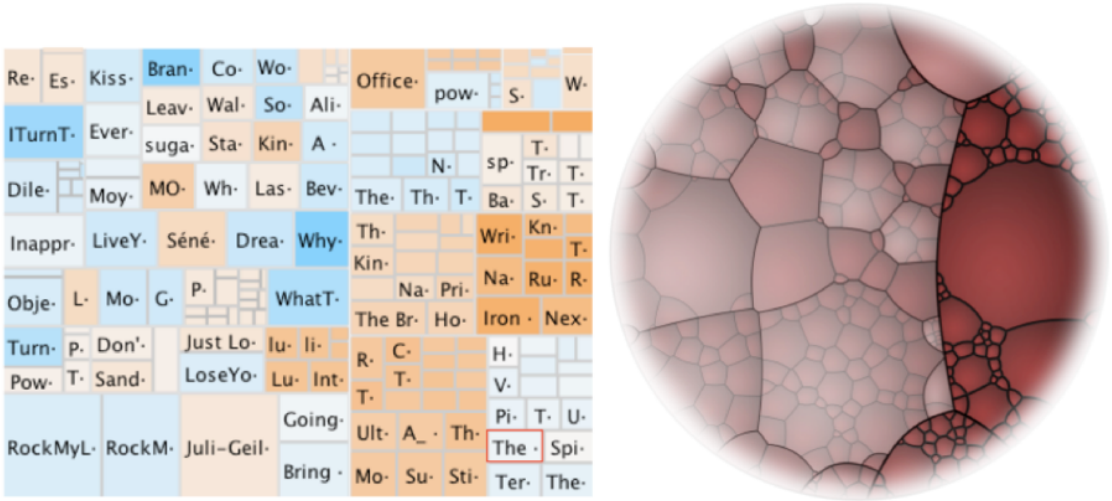


Fig. 3.1 a: Treemap visualization (derived from [40]) b: Voronoi Treemap (derived from [1])

optimized solutions. One of them is to use irregular shapes to encode elements so that each unit, class and component has unique shapes. This is the idea behind Voronoi Treemap [1] as shown in Figure 3.1(b). In addition, Voronoi Treemap uses color to distinguish different hierarchical levels, elements higher in the hierarchy are colored with darker red. There are also some attempts to extend Treemap to 3D visualization so that elements in each hierarchy can be put in different levels of elevation in 3D space (for example [4]).

3.1.2 Relationship Visualization

Compared to organization visualizations, visualizing the relationships between elements is more complex and accordingly requires more tricks. The reason is that the amount of relationships could be extremely large and there are more than one types of relationships such as inheritance, method call etc.

The node-link diagram is an intuitive way to display the relationships especially when the system is relatively small in size. In node-link diagram, software elements are depicted as nodes and the relationships between software elements are represented as links. UML, the most widely accepted software visualization technique, uses a node-link diagram to display the relationships between classes such as inheritance, generalization, associations, aggregations, and composition. Nodes in UML contains related information of classes such as variables and methods. As a result, there is a big amount of textual information when the system size grows which is space consuming and makes it highly prone to information overload [30].

The GoVisual UML provides a way to reduce the complexity. It restructures UML layout based on a set of aesthetic preference [22] to make sure that generalization relationships within the same class hierarchy follow the same direction. As we can see from Figure 3.2, (b) is more readable than (a).

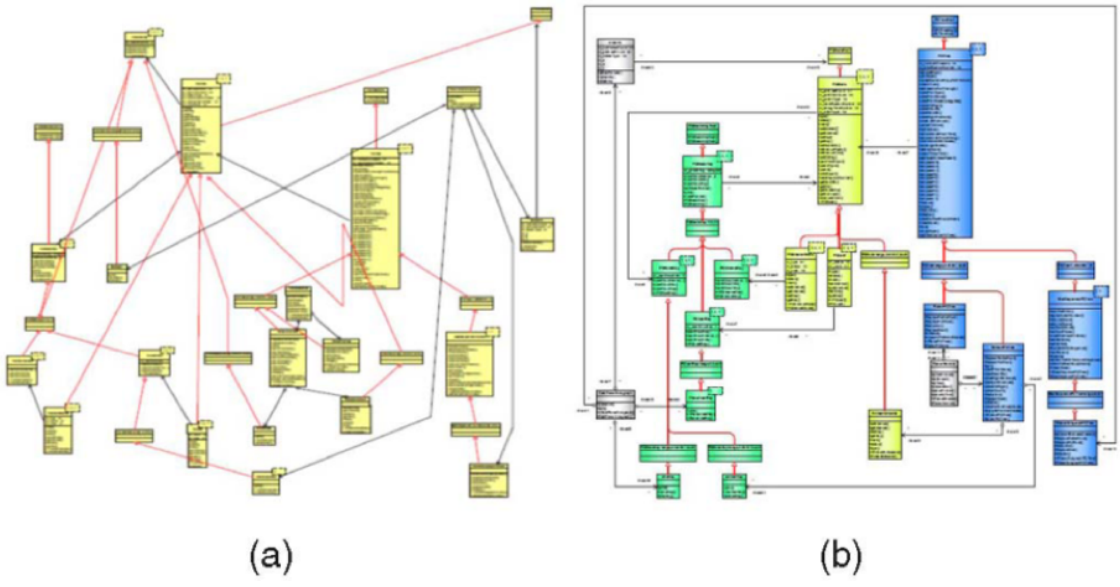


Fig. 3.2 (a) classical UML diagram, (b) GoVisual layout UML (derived from [11])

When the system's size grows, node-link diagrams can easily become clusters. An alternative way of visualizing relationships is the Dependency Structure Matrix [42] which is

a matrix with identical rows and columns. Each row and column represents an element in software, the number of dependencies between a row and a column is the value of the matrix. It's able to visualize the dependencies within a complex system in a simple and compact way. Dependency Structure Matrix's outstanding scalability enables it to display the relationships between elements at each level of hierarchy, but it doesn't support topological exploration.

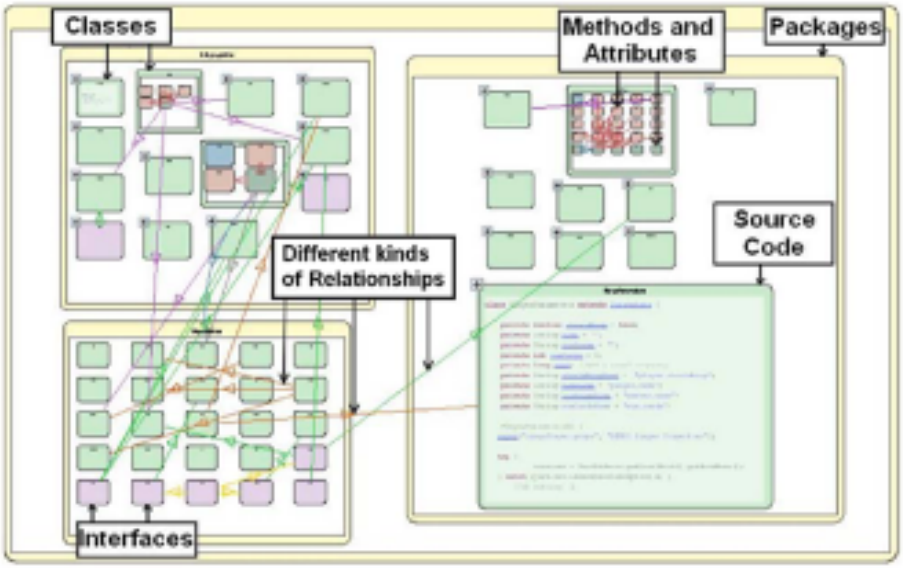


Fig. 3.3 Nested boxes with relationships from SHriMP (derived from [11])

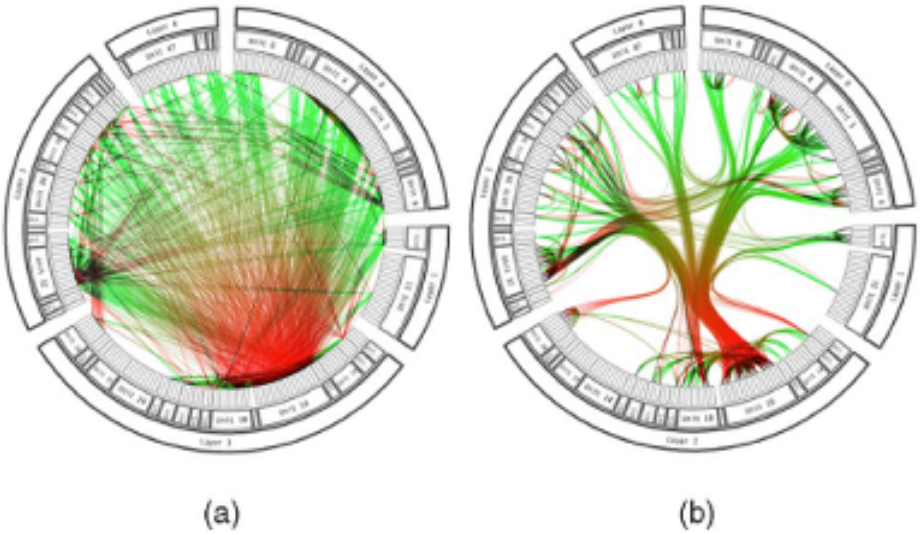


Fig. 3.4 Hierarchical edge bundling (a): Without bundling. (b): With bundling. (Taken from [26])

Prior research also came up with some techniques to visualize both hierarchical organization and the relationships between elements at the same time. SHriMP [43] is a multi-perspective visualization tool with the goal of helping developers understand Java programs. SHriMP combines a group of visualization techniques, for example, Treemap is one of them, and users are able to switch among them based on their needs. It can present a nested graph view to show the hierarchical information and then, relationships are displayed above the nested graph through directed lines as shown in Figure 3.3. The color of lines and boxes shows different types of relationships and elements respectively. Another important feature of SHriMP is that it also attaches source code and documentation in a single visualization so that developers can explore the architecture within the context of source code.

Holten [26] invented Hierarchical Edge Bundles (HEBs) which is a high level information visualization technique. HEBs puts all the elements on concentric circles and the depth of elements indicates their depth in the hierarchical tree. The concentric circles are also divided into several parts and each part represents a component. Then spline edges are used to show the relationships above the hierarchical visualization. HEBs uses colors to indicate relationship direction rather than arrows with green as source and red as target. As shown in Figure 3.4(a), elements in the bottom received a lot of dependencies from others. As the number of relationships increases, it can easily turn into a cluster. In order to reduce visual cluster, a bundling algorithm is then implemented to all the edges as shown as Figure 3.4(b). In this way, relationships are displayed in a well-structured way. This bundling technique can also be used on other organization visualizations so that it provides a way to combine organization and relationship visualizations.

3.2 Architecture Evolution Visualization

Having a global view of software evolution is deemed valuable, since it shows the history of the software, captures big changes and provides insights into the software's future development. In this section, we firstly introduce visualizations of organizational changes, then tackle the visualizations of metrics and relationship evolution.

3.2.1 Visualization of Organizational Changes

There definitely exists a requirement to visualize the hierarchical changes of software architecture over time. One approach is the work of Holten and Wijk [25] in which they proposed a technique to compare the software inheritance of two versions which in presented in Figure 3.5. Two versions are placed on two sides of the graph with the same nodes in two versions

placed opposite to each other. Then colors are used to show the differences between two versions, for example, red shading depicted the nodes presented in one version but not in the other. In order to reduce cluster, the bundling algorithm introduced in section 3.1.2 is adopted. In this way, the visualization shows the hierarchies of two versions and how they are related. This visualization technique has good scalability. It works well to compare two versions since the matching nodes are placed in similar relative position. However, only showing two versions at one time is a limitation when showing software evolution.

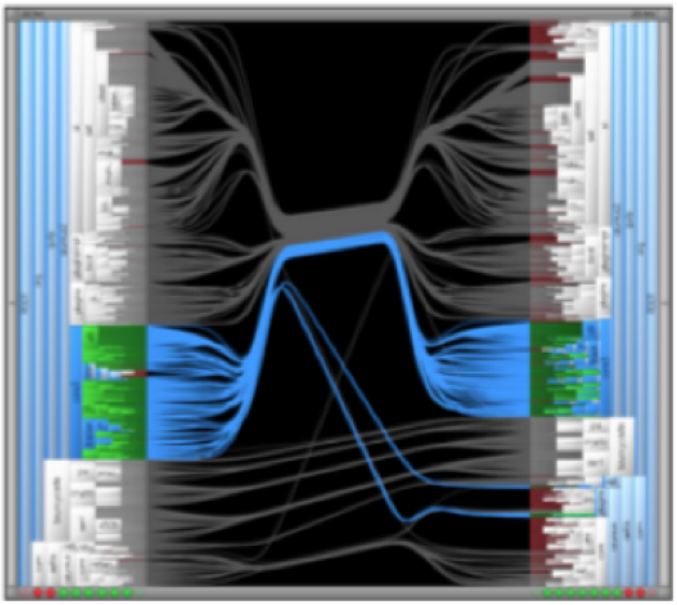


Fig. 3.5 Compare the hierarchical organization of two versions

(derived from [25])

3.2.2 Visualization of Relationship Evolution

Relationship change is an important part of structural change. However, there are only limited attempts to visualize the evolution of elements' relationships. One example is Gevol, which is a graph drawing technique for handling software evolution through visualization of large graphs with a temporal component [12]. Gevol collects software related information from the CVS version control system and then extracts inheritance graphs, call graphs, and control-flow graphs of the program. Furthermore, colors are used to depict the changes on these graphs. Figure 3.6 presents an example of inheritance graph evolution created by Gevol. It displays five versions of the inheritance graph. Initially, colors of nodes are assigned by its author (red, green, yellow) and the nodes without changes are progressively

fade to blue. This graph demonstrates the work of different authors and emphasizes the parts of code which remain unchanged. This approach is designed for large graphs with huge amount of nodes and edges, thus it mainly views relationships from the top, which leads to difficulties in identifying problems only concerning several nodes. In other words, fine-grained visualization techniques are still hard to find and this research attempts to fill the gap.

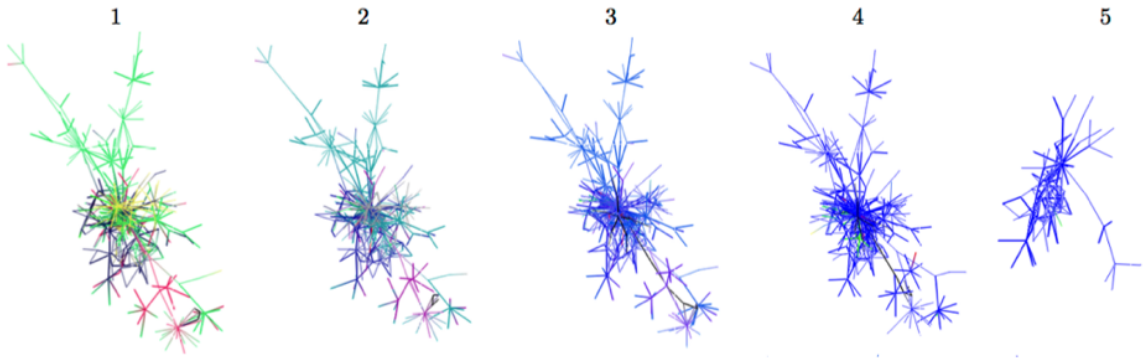


Fig. 3.6 An example of Gevol: inheritance graph evolution

(derived from [12])

3.3 Summary

Table 3.1 summarize all the visualization techniques studied from three aspects: the architecture features visualized, the way to encoding the features and pros and cons of each technique.

Techniques	Features	Encodings	Pros & cons
		Static aspects visualization	
Treemap	Units information(metrics), Hierarchy of the system	Nested boxes	**Pros:** Compact **Cons**: Unclear hierarchy level
Voronoi Treemap	Units information(metrics), Hierarchy of the system	Treemap using Irregular shape	**Pros:** Easy to differentiate every elements
3D Treemap	Units information(metrics), Hierarchy of the system	Treemap with polygons	**Pros:** Easy to differentiate hierarchy level
UML	The relationships between classes, Basic class information	Node-link diagram	**Pros:** Intuitive **Cons:** Limited to classes; risk of information overload
GoVisual layout UML	The relationships between classes, Basic class information	Well-structured UML	**Pros:** Less complex, clear layout **Cons:** Limited to classes; risk of information overload
Dependency Structure Matrix	The relationships between elements	Matrix view	**Pros:** Strong scalability **Cons:** Not intuitive
SHriMP	Both Hierarchical organization and relationship information	Add relationships above Treemap	**Pros:** Combines two types of information; connections to source code **Cons:** Cluster problem
HEBs	Both Hierarchical organization and relationship information	concentric circles with bundled spline edges	**Pros:** Combines two types of information; avoids cluster **Cons:** Less intuitive
		Evolution visualization	
Inheritance comparison	Hierarchical organization changes	Two sides comparison with bundled splines	**Pros:** Good scalability **Cons:** Only considers two versions
Gevol	Relationships (inheritance / call graphs / control-flow graphs) evolution	Large node-link diagram	**Pros:** Differentiate the work of different authors; strong scalability **Cons:** Coarse-grained

Fig. 3.7 Table 3.1 Summary of all the mentioned visualization techniques

Chapter 4

Prototype Design: Functionality

The main goal of this research is to develop a visualization prototype that can add value to both development and maintenance. Thus, we started with conducting prototype design that includes requirement analysis, functionality design and visualization design. Requirement analysis is implemented through interviews. After that, we translated the identified requirements to functionalities.

4.1 Requirement Analysis

The essence of visualization is not beautiful pictures, but the message delivered. We must make sure that the messages delivered by the visualization prototype are able to satisfy stakeholders' needs. Thus requirement analysis was conducted to identify different stakeholders' needs towards visualizations of software architecture evolution through interviews.

4.1.1 Interview Description

In order to collect requirements from different stakeholders, we'd like to select diversified interviewees. However, due to limited resources and time, we only selected stakeholders within SIG. In SIG, there are mainly three related roles: developers, managers of development and software consultants[1]. We tended to find candidates who are one of the three roles and have rich experience with software development or software architecture evaluation. Finally, we conducted three interviews: the first interviewee works as a developer and a development team manager at the same time, and sometimes, he also gets involved in consultancy work; the other two interviewees are software consultants. The three interviews

[1] In context of this research, software consultants perform static source code analysis to measure huge software portfolios and to derive hard facts from software to assess the quality and complexity of a system.

covers the opinions from all of the three roles mentioned before. We regarded the interviewees as the representatives of the roles. If there are contradictory requirements, we could have both. Therefore, we believe the results are sufficient enough to suit our purpose. In all cases, the interviewee is asked to answer the following two questions:

1. What level of details do you expect to see on software architecture evolution visualization?

2. What would be the visualization that helps you most in your daily work?

The summaries of these three interview are described below.

Interviewee 1 (developer, IT manager, consultant): Interviewee 1 answered the questions mainly from developers' perspective. But also mentioned the requirements of managers of development and consultants.

Visualizing the evolution of software architecture could definitely be helpful in my work. Imagine you get some information from the version control system to show how things changed, but it only shows file names and lines of code, which are low level changes. There are also changes on the class level and structure level which may not directly shown in version control system. Thus visualizing software architecture evolution could be useful.

As a developer, I expect both low level of details and architecture level of information. Sometimes, I need to work with a system which I'm not familiar with, for example, fixing bugs in existing system or integrating a new component. It would be better if I have some architecture visualizations that help me understand the system. I also need the visualization that can help me to review peer's work and then evaluate the impact of people's work at architectural level. Evolution visualizations enable me to identify large changes, for example, refactoring. In addition, visualizing the current state and future state helps me to figure out how to achieve our goals. For my daily work, I'd like to track the changes at both code level and architecture level, for example, call graph, dependencies, structural changes, technology and deployment.

As a manager, I want a visualization that can present our team's work, track architectural changes and support future decision. As a consultant, I need visualizations that facilitate architecture evaluation and identify new problems so that it also confirms that we fixed the problem for clients. Besides that, it's better to enable users to go back to the code because consultants need to show the problem of the code to clients, developers also have the demand to go back the code to fix bugs.

Interviewee 2 (consultant): *Software architecture is vague. Visualizations can externalize*

it, which is definitely helpful for not only my work but the whole industry. As a consultant, my work mainly lies in evaluating software architecture, either to identify where the risk is or to analyze the evolution to predict the future state of the architecture. For me, architecture is high level abstraction that prevents bad things, single class changes may be not interesting to me. Thus I expect to see the visualization at component level. I'm interesting to see all the changes at component level including component changes and dependency changes. As for dependencies, not only the dependencies between components but also the dependencies inside components will be considered. Besides, components' co-evolution is valuable. Co-evolution provides a lot of insights for architecture evolution, for example, component A and component B always change together, which may indicate there is hidden dependency between these two.

Interviewee 3 (consultant): *In my daily work as a software consultant, I mainly focus on component level and interfaces. I often get the task to evaluate the architecture of a client's system. The first step is to identify all the components. Based on the results, I look at how these components communicate with each other, in other words, I look at the dependencies between them. Generally speaking, for a system, there are usually more than one technologies, for example, the front-end of a web application could be developed in HTML and CSS, the back-end may be developed in Java. The dependency from front-end to back-end is cross technology dependency. Dependencies between components, especially cross technology dependencies are valuable. In addition, I always spend long time on finding call chains (for example, component A calls B, B calls C, C calls D). Thus I also expect a visualization of call chains.*

4.1.2 Conclusions of Interviews

From the three interviews, we reached the following conclusions:
For developers:
Requirements:

- Understand the software to facilitate maintenance or future development
- Track the changes on the code and evaluate the impact of changes
- Review peers' work
- Trace the changes back to the code
- Require mainly code level details and architecture level information is also needed

Architectural features expected: current state and changes on call graph, dependencies structure, technology and deployment.

For software consultants:
Requirements:

- Evaluate the software architecture
- Identify potential problems/risks
- Find existing problems
- Predict future state of software systems
- Present and explain their work to customers
- Analyze at component level and don't want to see the details at code level

Architectural features expected: Component changes, dependency changes, co-evolution, call chain, interfaces.

For managers of development
Requirements:

- Present the development team's work
- Identify problems in the system
- Facilitate decision making
- High level view of the architecture

Architectural features expected: global overview

4.1.3 Matching Required Features with Existing Techniques

In chapter 3, a set of existing visualization techniques are introduced. In this section, we'd like to match the existing techniques with the requirements identified through interviews. Basically, it's the match between the features required by users with the features provided by the introduced techniques. The results can be found in Table 4.1. The techniques only showing the static aspects of software can be further enhanced to visualize evolution by adding a time dimension as described in Section 2.3.

Table 4.1 The results of requirements and techniques matching(all the techniques are introduced in Chapter 3)

"_": no matching techniques, "\": not applicable

Stakeholders	Features	Existing techniques (static aspect)	Existing techniques (Evolution)
Developer	Current states and changes of call graph, dependencies	Dependency Structure Matrix, SHriMP	Gevol
	Organizational changes	Treemap, SHriMP	Inheritance comparison
	Technology	_	_
	Deployment	_	_
Consultant	Component changes	Treemap, SHriMP	_
	Dependency changes, call chains	Dependency Structure Matrix, SHriMP	Gevol
	Co-evolution	\	_
Managers of development	Global overviews	HEBs, SHriMP	_

Through interviews, we collected requirements from different stakeholders. However, there is no one-size-fits-all solution. An alternative way is to identify the requirements they have in common. As shown in Table 4.1, developers and consultants both require dependency visualizations and need information at component level. In addition, dependencies at component level can provide global overview which satisfies managers' requirements. Therefore, the feature we selected is component dependency which fits with multiple requirements. Furthermore, many cells on the table concerning with the evolution visualization techniques are empty, we'd like to fill the gap by highlighting the evolution of component dependency.

4.2 Requirements Translation

Based on the results of the interviews, we found that there are gaps between the requirements and the expected functionalities. Although we selected component dependency as the feature to be visualized, it still not clear what information can be delivered by the component dependency visualization and how the information reflects stakeholders' requirements. For instance, if there is a new dependency between two components, questions like "what's the influence of the new dependency?","is it good or bad?" will be asked. In order to fill the gaps, we try to find more information about evolution aspects of components' relationships.

Table 4.2 Functions and features

Functions	Features
Single version visualization	Components and dependencies among them
Two versions comparison	Changes on components and their dependencies
Evolution visualization	Evolution of components and their dependencies

Lightweight Sanity Check for Implemented Architectures (LiSCIA) is a lightweight architecture evaluation model focusing on the maintainability quality attribute of a software system which is invented by Bouwers (2013). LiSCIA is highly recognized by SIG experts and then used by SIG consultants as part of their daily practice. LiSCIA has been applied to a big number of systems.

LiSCIA provides a group of questions and actions linking to the questions to implement architecture evaluation from five perspectives: source group, component functionality, component size, component dependencies and technology. Source group refers to defining components and placing files into components, which is out of scope for this research. Component functionality indicates the way the sources are grouped into components which is also out of scope. Technology evaluation is related to evaluating the combination of technology within the system.

Evaluation of **component size** contains not only the size of a single component, but also the size distribution among components. The following features should be captured:

- Uneven size distribution. All components should have similar size.
- Big component size changes such as a component's size suddenly grows a lot. This could be an indicator of unwanted evolution and unwanted development effort.

Component dependency analysis intends to identify the wanted, unwanted and cyclic dependencies. These dependencies would be potential problems and without proper actions, they would result in higher maintenance expenses. The following features should be captured:

- Cyclic dependencies shown as loops in the dependency graph
- Unexpected dependencies which should be decided by the user
- The component has the most outgoing dependencies
- The component has the most incoming dependencies
- New dependencies and removed dependencies

4.3 Functionality

In this research, we selected the animated approach with a time dimension to visualize software architecture evolution. Versions are presented one after the other with animated transitions in between. By this means, it increases coherence between presented versions and therefore reduces cognitive gaps when switching from one version to another.

Evolution visualization is built on top of a single version visualization. Thus single version visualization is the foundation of this approach. While the essence of this approach is the animated transition from one version to another. Therefore, there are three functions: single version visualization, two versions comparison and evolution visualization. The features provided by each function are summarized in Table 4.2. More details about functionality will be provided in chapter 6.

Chapter 5

Prototype Design: Visualization

Visualization design builds the bridge from features to visualizations. In this chapter, we described visualization design which consists of basic visual encodings and layout design.

5.1 Marks and Channels

5.1.1 Theory

Marks and channels, the alphabet of visualization, are the basic elements to analyze visual encodings. Marks just like the blocks for a building and channels are the ways architects organize the blocks. Hence, analyzing marks and channels provide core information of visual encoding.

According to Tamara Munzner [38], a mark is a basic graphical element in an image which can be classified according to the numbers of special dimensions it requires. For example, a point is a 0D mark; an area is a 2D mark. A visual channel is a way to control the appearance of marks, independent of the dimensionality of the geometric primitives. There are two fundamentally different kinds of visual channels: identify channels which encode qualitative data and magnitude channels which are suitable for quantitative data. Channels are externalized by visual variables [3][23], which can be classified into seven groups: position, size, shape, value, color, orientation and texture. When encoding an attribute, there are two ways of deploying channels:

- One channel is assigned to encode a single attribute.
- Multiple channels are combined together to encode one attribute.

The first one is the most common case, which is easier to implement, while the second one is easier for the users to perceive the shown attribute.

The orthogonal combination of marks and channels provides a method to design and analyze visualization artifact. As shown in figure 1, the horizontal axis shows examples of marks: points, lines and areas. Accordingly, the vertical axis shows some visual variables as channels. Different combination of marks and channels defines different graph. There is no best combination which means people need to choose it based on the data type. The rightmost column of the chart gives suggestions on choosing channels. In general, this chart shows the basic idea of both analyzing and designing visual encoding through marks and channels. Over the years, the knowledge has been widely recognized by the cartographic society and further expended by researchers. For example, Morrison [37][23] put up with two new variables as channels: arrangement and saturation in 1974. Munzner [38] claimed that three-dimensional mark volume is possible in some 3D visualization, but they are not frequently used. In this thesis, theory of marks and channels is going to be used in visualization design and analysis.

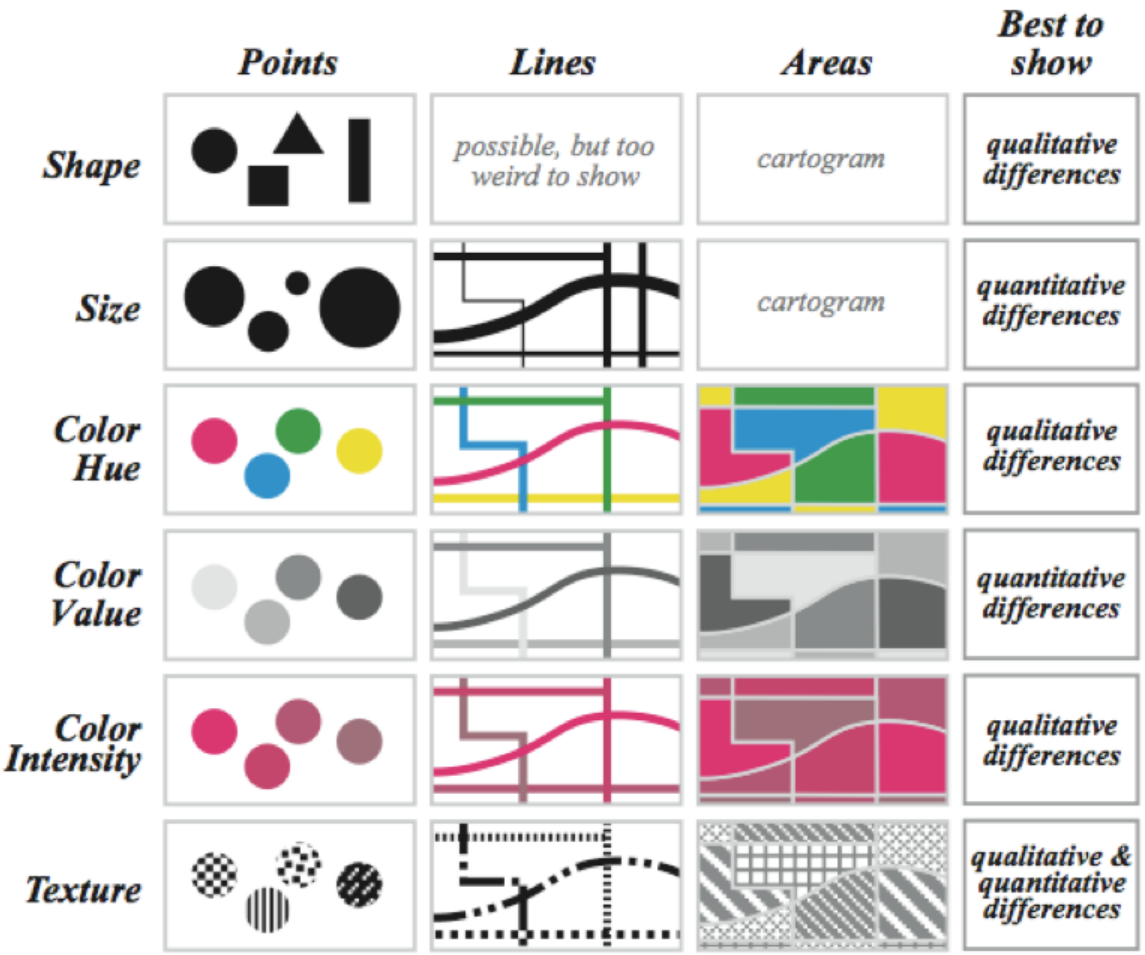


Fig. 5.1 Visual variable chart [32] showing different ways of visual encoding and possible application

Table 5.1 A simplified overview of data required by scenario 1

Version	Data	Format
Version 1	Components	Name: String; Size: Integer; Architecture layer: Position
	Dependencies	Source: String; Target: String; Size: Integer

5.1.2 Marks and Channels Design

As suggested by the theory, marks and channels should be determined by the dataset. Therefore, it's necessary to look into the data again. As discussed in the previous section, there are three scenarios in the visualization model: single version visualization, two versions comparison, and evolution visualization.

Scenario 1: Single Version Visualization

The data consists of two parts: components information and dependencies information, which is roughly summarized in Table 5.1.

Due to the generally agreed drawbacks of large 3D graphs, such as: object occlusion, cumbersome view adjustments, performance issues, as well as poor readability of 3D texts [35][13], we don't consider 3D marks in this research.

In terms of components, point marks are selected due to the following reasons:

- There are three key attributes: Name, Size and Architecture layer. Among them, architecture layer refers to components' position in the canvas. Components should be displayed in layered layout. The reasons are that layered architecture is one of the most widely used architecture style and for software using other architecture styles, there is also hierarchy among components. For instance, a component without outgoing dependencies should be put on the bottom. Since components have more than one attribute, we should choose marks that can be encoded by multiple channels
- Among these three attributes, Name is stored in text, which is not included in graphical elements
- Size and Position are both quantitative data, thus the mark represent components should support magnitude channels: size and position
- "The higher-dimensional mark types usually have built-in constraints that arise from the way they defined" [38]. For example, area marks have size attribute as part of their shape and are not size coded normally. While point marks can be size coded easily because they have no constraint on their area.

Table 5.2 A simplified overview of data required by scenario 2

Version	Data	Format
Version X	Components	Name: String; Size: Integer; Position: (x, y)
	Dependencies	Source: String; Target: String; Size: Integer
Version Y	Components	Name: String; Size: Integer; Position: (x, y)
	Dependencies	Source: String; Target: String; Size: Integer
Difference	Components	New components, Removed components, Components with size changes
	Dependencies	New dependencies, Removed dependencies, Dependencies number changes

Therefore, we choose point marks to show components and accordingly, choose size and position as channels.

As for dependencies, they show the relationship between components and mainly connection relationship in our case. So line marks are the most suitable mark for dependencies. Furthermore, line marks can be coded by orientation and color luminance to show the dependency direction and strength respectively. In our case, line marks are also referred to as connection marks.

Scenario 2: Two Versions Comparison

Compared to the first one, the differences between two versions are taken into consideration in this scenario. Table 5.2 shows the key data used here. The differences between two versions consist of components changes and dependencies changes, for each, there are three types of data: new elements, removed elements and altered ones. For the first two types, we can either visualize them through adding, removing marks or encode them with another channel. Considering that consulting human memory brings much more cognitive load than using visualizations as external representations, in this scenario, we choose the second solution: using texture and color as the extra channels to enhance human perception, which means instead of deleting marks directly form the image, we use dashed edges to code removed components and dependencies. With respect to altered components, there is no need to add more channels since point marks' size changes indicate component size changes. However, things are different for dependencies changes. Dependencies are coded by directed lines and apparently they are inadequate to indicate the dependency strength changes. Here we choose color hue because there are only two cases: stronger dependency and weaker dependency, and different colors enable people to differentiate these two cases easily.

Table 5.3 Summary of marks and channels

Scenario	Data	Mark	Channel
1	Component	Point	Size, Position
	Dependency	Line (connection)	Orientation, Color luminance
2	Component	Point	Size, Position
	Dependency	Line (connection)	Orientation, Color luminance
	New element	New point or line	Color hue
	Removed element	Point or Line	Texture
	Altered component	Point	Size
	Altered dependency	Line (connection)	Color hue
3	Component	Point	Size, Position
	Dependency	Line (connection)	Orientation, Color luminance
	New element	New point or line	Color hue
	Removed element	Removed point or line	\
	Altered component	Point	Size
	Altered dependency	Line (connection)	Color Luminance

Scenario 3: Evolution Visualization

Based on scenario 2, scenario 3 aims at visualizing the evolution from the first version to the last one. Basically, scenario 3 works in the similar way with the iteration of scenario 2. Therefore, there are only small changes of marks and channels.

Since the goal is to present the gradient changes, the channels for changes must be magnitude channel. Therefore, color hue is not applicable any more. In addition, removed elements should be directly removed from the graph rather than being encoded by texture.

In conclusion, Table 5.3 summarizes all the marks and channels used in this model.

5.2 Layout Design

Layout design is vital to the success of a visualization craft since layout determines the way to arrange marks and channels spatially. If the layout is not well designed, the message delivered by the visualization could be unclear or misunderstanding.

Since our dataset is made up of components and their relationships, it can be regarded as a small network. When arranging networks, the most common visual encoding idioms are node-link diagrams and adjacency matrix. Table 5.4 shows the comparison between these

Table 5.4 Node-link diagram and adjacency matrix comparison

	Node-Link diagrams	Adjacency Matrix
Encode	Point marks for nodes, Connection marks for links	Area marks in 2D matrix alignment
Task	Explore topology	Display key value
Scale	Limits on the amounts of nodes and links	No limits on link numbers, Little limits on node numbers
Readability	Easy to interpret	Not easy to interpret

two idioms. Node-link diagram uses point and connection marks to show the topological structure. For small networks, node-link diagrams are extremely intuitive for solving abstract tasks, such as finding cycles in the graph. Adjacency matrix is usually derived from tables and denotes nodes as the keys and link status as the values. In this way, it's not restricted by both scale and density of the network.

In this research, evolution is the focal point. Node-link diagrams are better to show the topological changes while adjacency matrix is better to show the value changes. Therefore, we use node-link diagrams here, rather than matrix view. As a result, components in the dataset will be nodes in the network and dependencies are going to be depicted as links.

Force-directed layout which is firstly mentioned by Eades [28] is one of the most widely used node-link diagram. It places nodes and links using the analogy of physical springs with the aim of minimizing node overlapping and link crossing. In the simulated system, nodes and links imitate the behavior of charged balls and connecting springs. In other words, nodes push away from each other while links draw their endpoint nodes closer to each other. In this way, a system consisting of physical objectives with an equilibrium configuration is created to make sure that all the nodes are spread well in the graph.

There are different algorithms to build the equilibrium configuration, for example, in D3 library, Verlet integration is used. Basically, Verlet integration is a numerical method used to integrated Newton's equations of motion [48]. For each node, it defines two main variables: its position and its velocity. However, in implementation phase, velocity is implicitly given, only its current position and previous position are stored, and the positions will be updated in a fixed time step. Therefore, all the nodes are placed randomly at the beginning and then the layout algorithm will iteratively adjust their positions according to the pushing and pulling forces to gradually improve the layout.

Force-directed layout has some obvious advantages. For example, "graphs drawing with this layout tend to be aesthetically pleasing, exhibit symmetries, and tend to produce crossing-free layouts for planar graphs" [45]. Furthermore, it's relatively easy to understand and explain at a conceptual level [38]. However, there are several major defects that need

to be considered as well. Typically, force-directed layout won't use spatial position to encode attributes, since the layout algorithm uses the position directly to avoid distracting artifacts such as node overlapping. However, from user's perspective, the positions of nodes can convey a great deal of information. Thus users will try to figure out the meaning of the implicitly chosen position, which is misleading. In our case, there is a hierarchy of components which means the algorithm need to be adjusted so that it can minimize distractions and support position encoding at the same time.

The adjustment is based on the weights of components. There are two ways of calculating weights of components.

- Using incoming and outgoing dependencies value of a component to calculate its weight
- Using the number of components dependent on the component, and the number of components on which the component depends to calculate its weight

In both cases, the calculating algorithm is:

- If incoming value is 0, the component is placed on the top
- If outgoing value is 0, the component is placed on the bottom
- Otherwise, use the difference between incoming value and outgoing value as the weight, the bigger the weight is, the lower the node is placed.

The first method adds up all the dependency values and puts the component with biggest value on the bottom, while the second method puts the component with the most callers on the bottom. In this way, all links are designed to toward the same direction and any link towards the top may cause cyclic dependencies. By adjusting the algorithm, the horizontal position of a node is determined by the force-directed layout, the vertical position of a node is determined by the adjustment algorithm introduced above.

Another major disadvantage of force directed layout is that it has limits on its size and the link-node density. As soon as the size of the graph or the link density increases, "force directed layout is facing occlusion problems due to links overlapping" [20]. According to Tamara Munzner [38], the idea is that link-node density should be smaller than four and nodes could be dozens or hundreds. In our case, the influence is limited. As suggested by a survey conducted by Bouwers [8], the most common number of components for a system is close to 8. And there is no strong correlation between the number of components within a system and the system's size which is represented by lines of code.

In order to further minimize the effects, user interactions are used. Force directed layout support clicking and dragging nodes to show more information and minimize distractions. In addition, functionalities such as search and selection can also contribute. More details about user interaction will be provided in next chapter.

Chapter 6

Visualization Prototype Implementation

To solve the raised research question 3, a prototype was built based on the design introduced in Chapter 4, which fits multiple stakeholders' needs of architecture evolution visualization. In this chapter, the implementation and final results of this prototype are described.

6.1 Data

6.1.1 Data Collection

In order to implement the design introduced in chapter 4, architectural information of software is needed. However, getting architectural information is not an easy task since implemented architecture is not always in compliance with its documented design. Moreover, along with software evolution, architecture erosion happens inevitably and in some cases, even architects and developers involved don't know everything about the architecture. Therefore, both documentation and input from stakeholders are not reliable. In this case, an alternative way to abstract implemented architecture information is to analyze the source code.

The dataset used in this research is derived from the results of SAT, which is a source code analysis tool developed and used by the Software Improvement Group ("SIG"). Based on source code analysis, SAT measures the maintainability of software systems, and provides the results via a benchmarked star rating. The data captured by SAT includes technology usage, software metrics and architectural information etc.

In this research, architectural information and several related software metrics are chosen to form the visualization dataset. Finally, the dataset is accessed as JSON served over HTTP.

6.1.2 Data Description

The dataset is constructed in layered structure with six layers: system, snapshot, technology, architecture levels, dependency information, software metrics as shown in Figure 6.1.

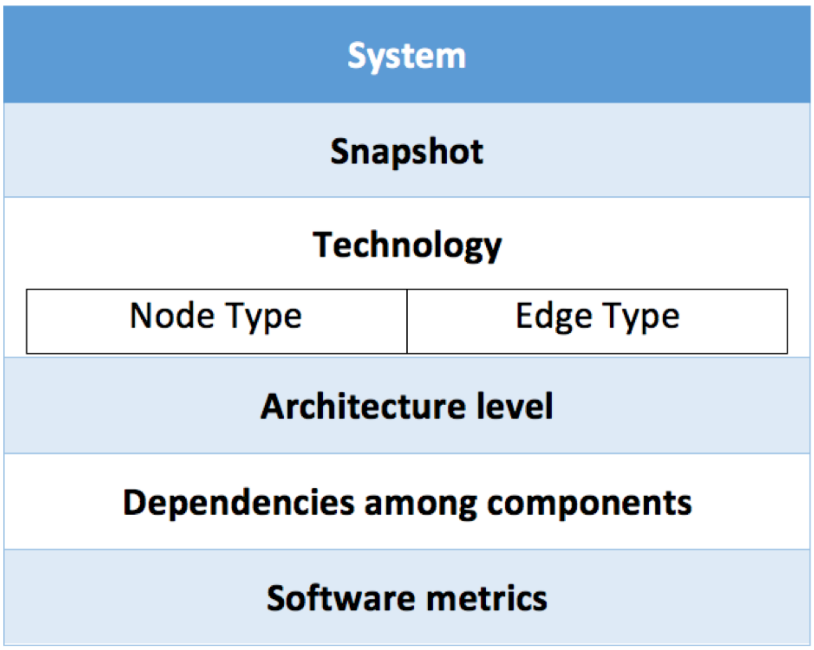


Fig. 6.1 Layered structure of dataset

System: The dataset consists of 35 open source systems and several SIG internal systems; each time users are asked to choose one system.

Snapshot: snapshot is the state of a system at a certain point in time. SAT implements regular inspection of the software systems, and each time a snapshot is generated and stored in the database. Hence, snapshots indicate the way the systems evolve.

Technology: technology refers to programming language such as Java, C#, JavaScript, etc. At this layer, for each technology, there are two branches: node type and edge type. Edge type stores all the dependency types, for instance, call dependency. Node type directs to the next layer.

Architecture Level: Nodes can represent elements at different architecture level, such as units, classes, components. In context of this research, only components are considered.

Dependencies among components: this layer stores dependencies information which consists of snapshot date, source component, dependency type, dependency number (weight) and target component as shown in figure 6.2.

Software metrics: Two software metrics in the dataset are used in this research: Lines of Code and Component Independence.

- Lines of Code: Measure the size of a software by counting the number of lines in source code. The value of Lines of Code is used to encode nodes' size.
- Component Independence: Measures how cross-linked a component is. Dependencies of a component can be divided into four categories: Hidden, Incoming, Outgoing and Throughput, as illustrated in figure 6.3 [8]. Hidden dependency means the elements inside the component depend on each other. Incoming dependency refers to the dependencies form elements outside the component to elements inside the component. Reversely, outgoing dependency refers to the dependencies from elements inside a component to outside elements. Throughput dependency means the elements inside the component both have incoming and outgoing dependencies. This metrics quantifies the percentage of code in hidden and outgoing category which is not directly depended upon by outside elements. The higher the percentage is, the higher the rating of Component Independence is [8].

Snapshot

1
2013-11-19 00:00:00 +0000 UTC

Dependencies

Source	Type	Weight	Target
hadoop-maven-plugins (6858)	java.call	29	hadoop-maven-plugins (6858)
hadoop-tools (6875)	java.call	315	hadoop-yarn-project (7526)
hadoop-tools (6875)	java.call	565	hadoop-mapreduce-project (5364)
hadoop-mapreduce-project (5364)	java.call	1076	hadoop-yarn-project (7526)
hadoop-yarn-project (7526)	java.call	1762	hadoop-common-project (1748)
hadoop-tools (6875)	java.call	1802	hadoop-common-project (1748)
hadoop-mapreduce-project (5364)	java.call	3729	hadoop-common-project (1748)
hadoop-tools (6875)	java.call	3736	hadoop-tools (6875)
hadoop-hdfs-project (3544)	java.call	3852	hadoop-common-project (1748)
hadoop-mapreduce-project (5364)	java.call	10166	hadoop-mapreduce-project (5364)
hadoop-yarn-project (7526)	java.call	10327	hadoop-yarn-project (7526)
hadoop-common-project (1748)	java.call	11667	hadoop-common-project (1748)
hadoop-hdfs-project (3544)	java.call	16078	hadoop-hdfs-project (3544)
hadoop-tools (6875)	java.extends	1	hadoop-yarn-project (7526)
hadoop-tools (6875)	java.extends	16	hadoop-common-project (1748)
hadoop-tools (6875)	java.extends	36	hadoop-mapreduce-project (5364)
hadoop-mapreduce-project (5364)	java.extends	70	hadoop-common-project (1748)
hadoop-mapreduce-project (5364)	java.extends	70	hadoop-yarn-project (7526)
hadoop-yarn-project (7526)	java.extends	90	hadoop-common-project (1748)
hadoop-hdfs-project (3544)	java.extends	116	hadoop-common-project (1748)
hadoop-tools (6875)	java.extends	159	hadoop-tools (6875)
hadoop-hdfs-project (3544)	java.extends	306	hadoop-hdfs-project (3544)

Fig. 6.2 Dependency information.

This is the first snapshot (Date: 2013-11-19)

6.2 Prototype Building

In this section, the technical implementation and major functionality of the prototype are described.

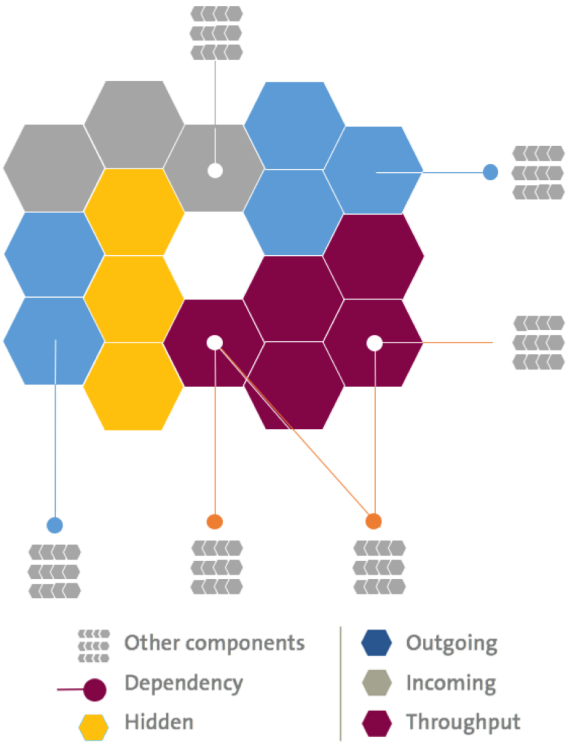


Fig. 6.3 Dependency types

Four categories of dependencies: Outgoing, Throughput, Hidden and Incoming [8].

6.2.1 Tools

Increasingly, the web is being used in information visualization field. Given this trend, the web has naturally progressed as a source of information as well as an underlying delivery mechanism for interactive information visualization [41]. According to Michael Bostock et al., one of the great successes of the web as a platform is the seamless cooperation of multiple technologies: HTML for page content, CSS for aesthetics, JavaScript for interaction, SVG for vector graphics, enabled by a shared representation of the page called the document object model (DOM) [7]. This interlink associates data, visual representation and users creates new opportunities for visualization. Therefore, in this research, web based visualization approach is adopted.

D3, a JavaScript library that helps to visualize documents based on data, is one of the most widely used web based visualization technology. It uses domain specific language, provides a declarative framework for mapping data to visual elements since it binds data directly into DOM instead of imposing a toolkit-specific lexicon of graphical marks [7]. In this way, D3 allows users to focus on their specific application domain rather than spending time on procedures, and guarantees great control over final visual results.

In this research, we have the following requirements for the visualization tool and D3 satisfies all the requirements:

- **Functionality.** The tool should provide rich visualization functionality to support users. D3 provides great flexibility since it allows users to manipulate any part of DOM. D3 provides a big variety of SVG elements and layouts which lead to rich visualization possibilities. And it supports dynamic behaviors for animation and user interaction.

- **Flexibility.** The tool should support customized design to fulfill all the requirements. D3 not only provides plenty of reusable charts but also enables users to design the graph and add customized functions as they like. In addition, D3 is extremely fast and works smoothly with large data set.

- **Usability.** The tool should be easy to use and support developing usage so that the prototype can be further developed. D3 supports multiple data formats such as JSON, CSV and debugging mode which increases its usability as a developing tool.

- **Open source.** Considering the fact that we have limited budget, we should choose a open source tool. D3 is an open source tool that is easier to acquire and use during researches. Furthermore, a lot of people are contributing to maintaining and further developing D3 on GitHub, making it more reliable to use.

- **Integration potential.** The selected tool should have the potential to be integrated with SIG internal tool so that it can be used in the industry in the future. SIG internal tools uses D3, so it brings big potential that the prototype developed in this research can be integrated into SIG existing tools in the future.

Therefore, D3 is selected as the tool.

6.2.2 Functionality

In this section, the functionality of the prototype is described in detail. In chapter 4, the information that needs to be delivered and the visual encoding scheme are discussed and

Table 6.1 Summary of information and related visual encodings

Scenario	Information	Visual encoding
Visualize Single Snapshot	Components and dependencies among them	Node-Link diagram with point marks as items and connection marks as links, all marks are encoded by multiple channels (See chapter 5.1.2)
Compare Two Snapshots	Changes on components and dependencies	Changes indicated by channels such as color, size, etc
Evolution	Architecture evolution	Animated dependency graph

these two determine the functionality of the prototype. As discussed, the prototype provides three main functions: visualizing component dependencies, visualizing the differences of component dependencies between two snapshots and visualizing the evolution of components and dependencies. Table 6.1 summaries the "what" and "how" for each scenario.

Visualizing Component Dependencies of a Single Snapshot

As shown in figure 6.4, the drop down lists in tool bar enable users to specify the system, the snapshot and technology they want to analyze. The purposes of the buttons are listed below:

- Start: Start to draw the graph.
- Previous: Jump to the previous snapshot. No action if the current snapshot is the first one.
- Next: Jump to the next snapshot. No action if the current snapshot is the last one.
- Cycles: Highlight cyclic dependencies on the graph. Since red color is also used in other scenario, this function only works in this scenario to avoid confusion.

We show the Hadoop (https://github.com/apache/hadoop) open source project as an example. The result of the visualization is illustrated in figure 6.5. On the graph, circles represent components in the software with their size indicating the size of components. The names of components are displayed in text. Links represent dependencies between two components. The color luminance of the links is determined by links' weight, in other words, the number of dependencies. The darker the link is; the more dependencies it has. As we can see from the graph, the link from hadoophdfs-project to hadoopcommon-project is the strongest one in this system. As described in previous chapter, a component's position is determined by component independence metric as described in chapter 4. As a result, all the

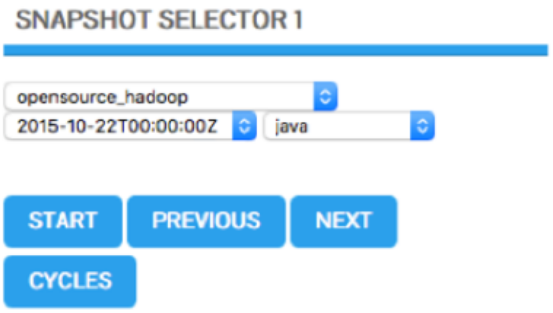


Fig. 6.4 Tool bar in first scenario

Users can choose system, snapshot (denoted by the date of the snapshot) and technology (Java, JavaScript, etc).

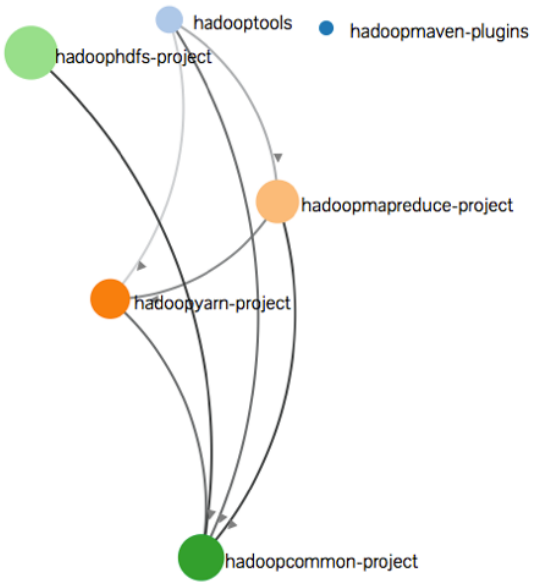


Fig. 6.5 Component dependency graph of system Hadoop

(Snapshot Date: 2013-11-19)

Table 6.2 Architectural changes and their visual encodings in scenario 2

Architectural changes	Components' changes	New components	New nodes
		Removed components	Nodes with dashed strokes
		Existing components with size changes	Nodes size change
	Dependencies' changes	New dependencies	New links
		Removed dependencies	Dashed links
		Existing dependencies with weight changes	Links color changes: Orange: stronger dependencies Blue: weaker dependencies

components are displayed in three layers – source nodes are at the top layer, sink nodes are at the bottom layer and nodes with both incoming and outgoing links are in the middle layer.

Apart from the graph, there is an information dashboard exhibiting basic conclusions as shown in figure 6.6. It contains the date of the snapshot, total number of components and dependencies in the chosen snapshot, the component with most incoming dependencies (biggest target on the figure) and the one with the most outgoing dependencies (biggest source on the figure).

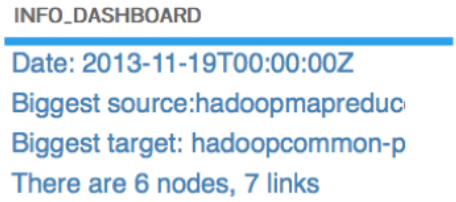


Fig. 6.6 Information dashboard

Visualizing the Differences

This function aims at helping stakeholders capture the architecture changes between two snapshot, so that the developing work during the time period can be displayed and evaluated afterwards. All the changes visualized are listed below in Table 6.2.

The visualization for the second scenario is shown in figure 6.7. Based on the visualization, we have the following findings:

- There are three new components with new dependencies: g, f, h

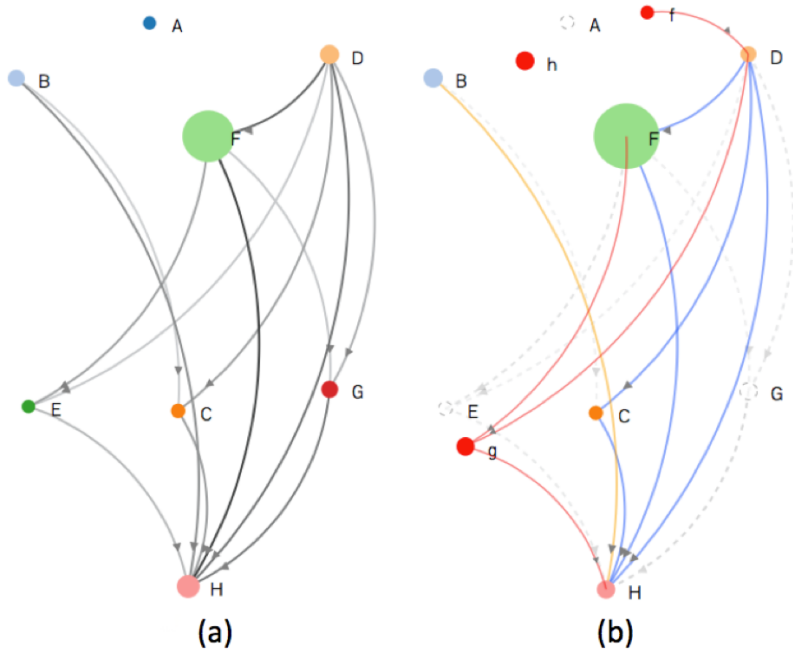


Fig. 6.7 Visualizing the changes between two snapshots

(a) displays the original snapshot, (b) displays the comparison.

- Component A, G, E and related dependencies have been removed
- Dependency from B to C has been removed
- There is no new dependency between existing components
- Dependency from B to H has become stronger
- Dependencies from D to F, D to C, D to H, F to H and C to H have become weaker
- Component F's size has grown

Visualizing the Evolution

Visualizing software evolution helps explain the current state of the software and depict important changes such as refactoring and architecture migration. In this way it can provide plenty of information such as the modules that have been under heavy maintenance, or, on the contrary, barely touched along their entire evolution [11][16][24][17]. Visualizing the software architecture evolution in this research means adding a time dimension to the single snapshot representation, architecture in snapshots are presented one after another using animation. Therefore, we animate the transition between each two consecutive snapshots

from the first one to the last one. In this way, the cognitive gaps are reduced and coherence between snapshots are increased when transferring from one snapshot to another. Figure 6.8

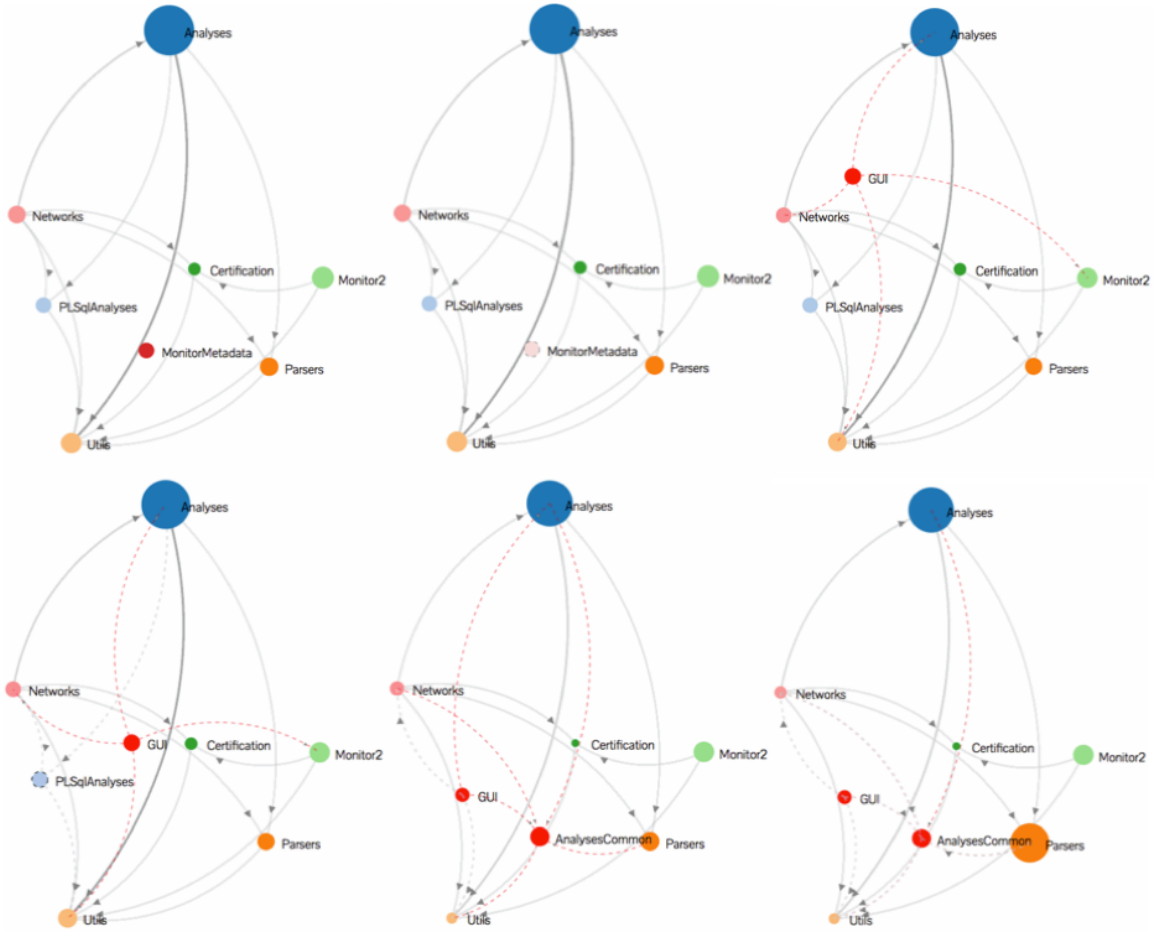


Fig. 6.8 An example of evolution visualization

illustrates a period of the evolution process of one Java system. It provides an overview of architectural changes over time. The following facts can be found during the evolution:

- Component MonitorMetadata and PLAqlAnalyses and related dependencies have been removed successively
- Component A, G, E and related dependencies have been removed
- Component GUI and AnalysesCommon with related dependencies have been added consecutively
- Component Parsers has become bigger
- Dependencies from GUI to Analyses and Monitor2 have been removed

- Dependencies from D to F, D to C, D to H, F to H, C to H have become weaker
- Dependency from Analyses to Utils has become weaker

User Interaction

User interaction plays an important role in this visualization prototype. According to Ji Soo Yi et al., a visualization usually has two essential parts: representation and interaction [50]. Representation mainly refers to mapping data into graphical elements and conveys the vast majority of the information. Interaction is concerned with the communication between users and visualization tools. Based on visualization requirements and human visual powers, representations have to make tradeoffs between various factors, which causes intrinsic limitations within representation [15]. While interaction can allow alternative representations to enhance the expressiveness. Therefore, Interaction can help users to achieve better understanding of the information delivered by visualization.

Ji Soo Yi et al. [50] put up with seven general categories of interaction techniques widely used in information visualization field: 1) Select, 2) Explore, 3) Reconfigure, 4) Encode, 5) Abstract/Elaborate, 6) Filter, and 7) Connect. In this prototype, Select, Explore, Filter, Abstract/Elaborate and Connect techniques are used.

Select & Filter: The prototype allows users to select a node and then only related nodes and links will be shown (Figure 6.9), So that users' attention can be gathered to one component. When the number of components and dependencies grow, the expressiveness of force-directed layout decreases a lot. Then this function helps to minimize the effects.

Explore: The prototype allows users to analyze different snapshots and technologies to explore the dataset (Figure 6.10). In this way, only a subset of data will be visualized and users can focus on one snapshot each time.
In addition, the force-directed layout has some interactive features as well. For example, users can drag the nodes to avoid nodes' overlapping and links' crossing so that more information can be displayed.

Abstract/Elaborate: Abstract and Elaborate techniques mainly control the level of details shown to the users. In this prototype, through mouse events: "right click" and "move over", more details will be displayed (Figure 6.11(a)). In response to "right click" event, a pie chart displaying the percentage of four types of dependency (Hidden, Incoming, Outgoing, Throughput dependencies) will appear(Figure 6.11(b)). As for "move over" event, the size of selected element will appear (component size or dependency number).

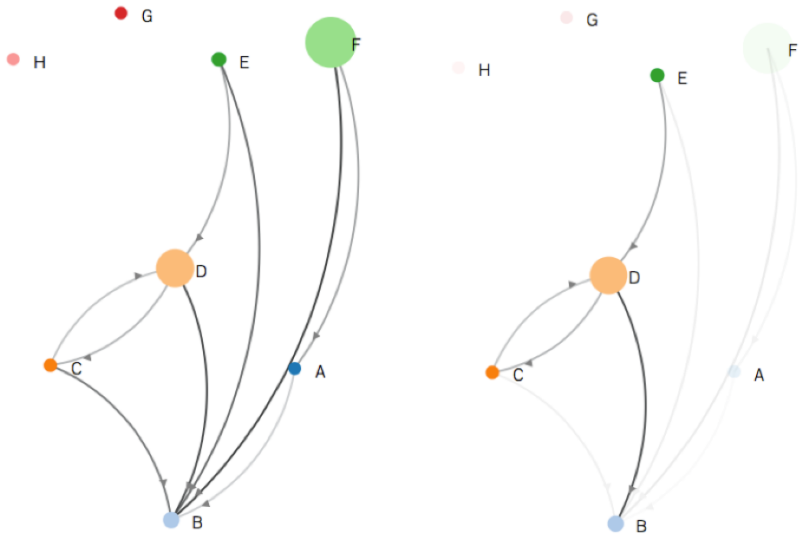


Fig. 6.9 Select and Filter

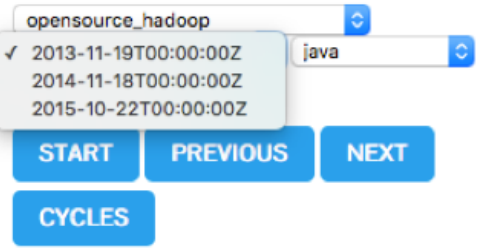
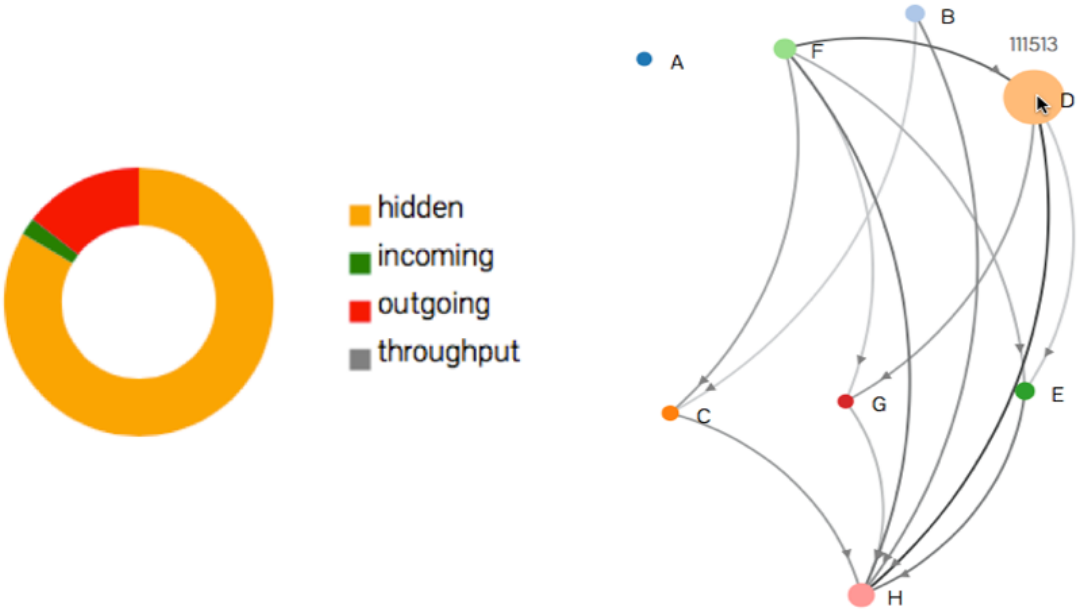


Fig. 6.10 Explore the dataset



(a) Elaborate function: right click (b) Elaborate function: move over

Fig. 6.11 Elaborate

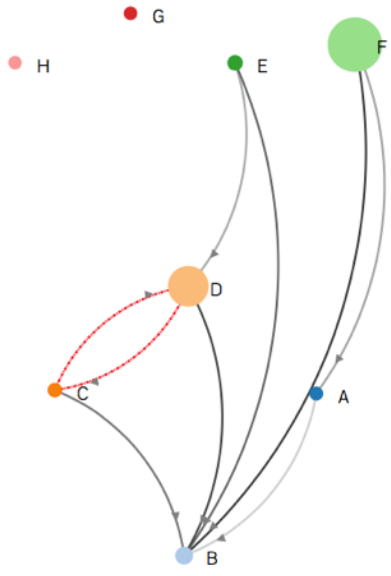


Fig. 6.12 Connect

The cyclic dependencies between component C and D are marked as red

Connect: Connect refers to interaction techniques that highlight relationships between items. This prototype provides a function to identify cyclic dependency through interaction, as shown in figure 6.12. Connect techniques are used here in order to avoid showing too much information at one time.

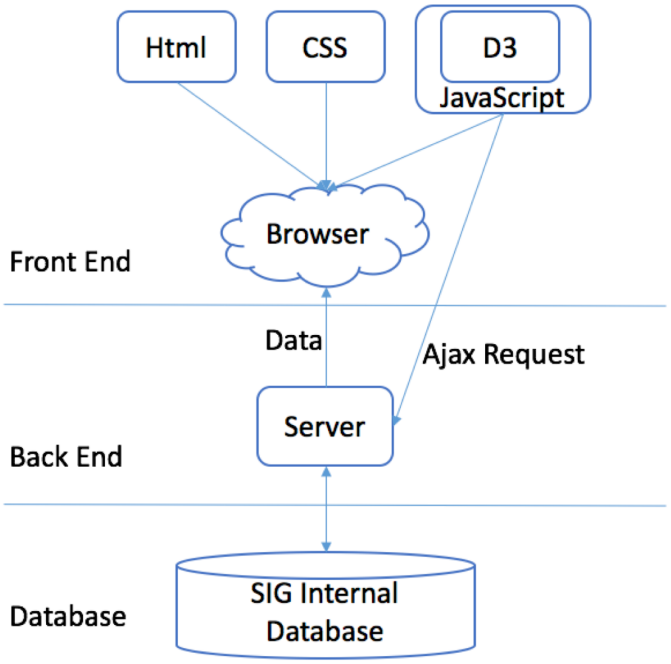


Fig. 6.13 Prototype architecture

6.2.3 Prototype Structure

This prototype is comprised of three components: front end, back end and database. The front end is developed using Html, CSS and JavaScript. The JavaScript library D3 is the core to implement visualization. The front end uses D3 mechanism calling Ajax request to get real time data from backend to browser. The Web based architecture of the prototype is illustrated in Figure 6.13.

Chapter 7

Validation

In previous chapter, we proposed a way to visualize implemented software architecture evolution and developed a prototype. In this chapter, using three case studies on different systems, we attempt to explore and interpret the applicability and usefulness of this prototype.

7.1 Validation Design

In order to validate the visualization prototype, three case studies are implemented. Validation consists of the following steps:

- **System screening:** Due to the limitation of scalability of force-directed layout, we only looked at small and medium sized systems with fewer than 30 components. Among these systems, we selected systems with more than five snapshots so that there is enough evolution information to show.
- **Send out invitation letter:** After system screening, we sent out invitation letter to stakeholders and invited them to participate in the validation. The invitation letter can be found on Appendix A.
- **Apply the visualization prototype to selected systems:** After receiving feedback from stakeholders, we started to get the related data from SIG internal database. When required data was ready, we applied the prototype to the system.
- **Collect stakeholder feedback:** On this stage, we conducted an interview with stakeholders to collect feedback.

During the interview, we showed stakeholders the visualization results, guided them to explore the prototype and reviewed the evolution of their system at the same time. The following categories of questions are discussed in the first two interviews:

- **Usefulness**
  - To what extent, the prototype helps you to understand the system?
  - Can the visualization showing architecture evolution help you with finding potential problems?
- **Applicability**
  - Do you like to use this prototype in your daily work? If so, how do you use it?
  - Compare with static pictures, which approach do you like more?
- **General comments and improvements**

As for third interview, we intend to solve a specific problem within the system.

The potential problems mentioned above consists of the following as we discussed before in chapter 4:

- Cyclic dependency
- Unevenly distributed components
- The component has most incoming and outgoing dependencies
- Unexpected dependencies
- Big size changes of components
  - Components keep growing/shrinking
  - Components keep growing while related components keep shrinking

7.2 Case Study 1

7.2.1 Background

Company A develops and provides software to check the quality of public constructions design artifacts such as railway designs. Their system that is developed in Java has a long history and has experienced several big architectural changes. In their early stage, they used layered architecture and had several work branches. Then a shared component was introduced and this component quickly became big and complicated. So they tried to split it up and as a result, there were more components and dependencies. Moreover, since new

functionality was introduced, their architecture quality decreased. And then in early 2016, they started to work on refining their architecture. This system has 43 snapshots ranging from 2007 to 2016. During the evolution, there are 22 nodes and 73 links at most.

7.2.2 Interview

We implemented an interview with two stakeholders from company A: an architect and a developer. Both of them have been working in company A for a long time and participated in the development of the system.

The first part of the interview was to show the prototype to them. When we were viewing the animated evolution, they were able to find out the following facts:

- At the beginning, the layered architecture changed a lot
- The growth of the large shared component
- After the introduction of the shared component, there were many new components and dependencies coming out
- Around October 2015, there were large changes visible, caused by the change in names of the components due to the migration of SVN to Git as their version control tool
- Around the beginning of 2016, they can see the clear layering in the architecture
- Around the beginning of 2016 the architecture already became more clear with better component distribution and less dependencies because they were progressing with migrating the architecture

In conclusion, the prototype helped them to review the development history. During the review, they can easily recognize big architectural changes and compare their new architecture with old one. It helped them confirm that they are doing the right thing in regards to migrating the architecture.

With the help of the visualization prototype, they identified several architectural violations.

Unexpected dependencies: During the interview, the architect pointed out that the dependencies from component m to I and component h to I were unexpected dependencies (see Figure 7.1). In the graph, all of the three component are positioned on the top. It's because they have a lot of outgoing dependencies and a little or zero incoming dependencies which means these components mainly depend on other components but

are rarely dependent by other components. Hence, there is high possibility that they are applications. This finding was proved by the architect and he said there shouldn't be dependencies between two different applications.

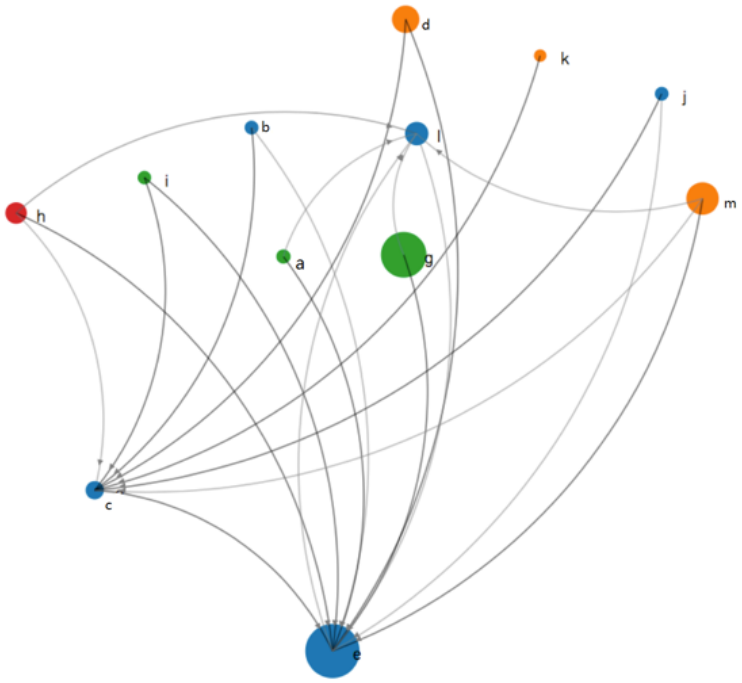


Fig. 7.1 Unexpected dependencies

Cyclic dependency: As shown in figure 7.2, cyclic dependency was identified by the prototype atomically. As said by the architect, at that time, they spent relatively long time to find the cyclic dependency between component A and D and it was quite expensive to remove it at that time.

In terms of applicability, they see the most value for this tool in:

Communication: The prototype can be used as a communication tool towards both technical and non-technical stakeholders. For example, the visualization helps architects and developers report their work to non-technical managers, since it externalizes technical work with graphs and animations that can be perceived by non-technical people. Moreover, compared to traditional visualization with static graphs, they think this prototype is more attractive. For technical communication, there are multiple developing teams working on the system, the communication across teams are quite difficult. While this visualization prototype enables them to review peer's work from top view, which can facilitate the cross-team communication.

Monitoring: The prototype can be used as a daily monitor tool for architects and developers. It can help to track architecture violations as soon as they appear. As said by the interviewee, cyclic dependency and the strength of dependencies are extremely important to them.

In conclusion, as a communication tool, the prototype delivers the existing information in a more accessible way and supports collaborative work. As a monitoring tool, the prototype indicates new information such as helping technical to identify existing anomalies and find potential problems.

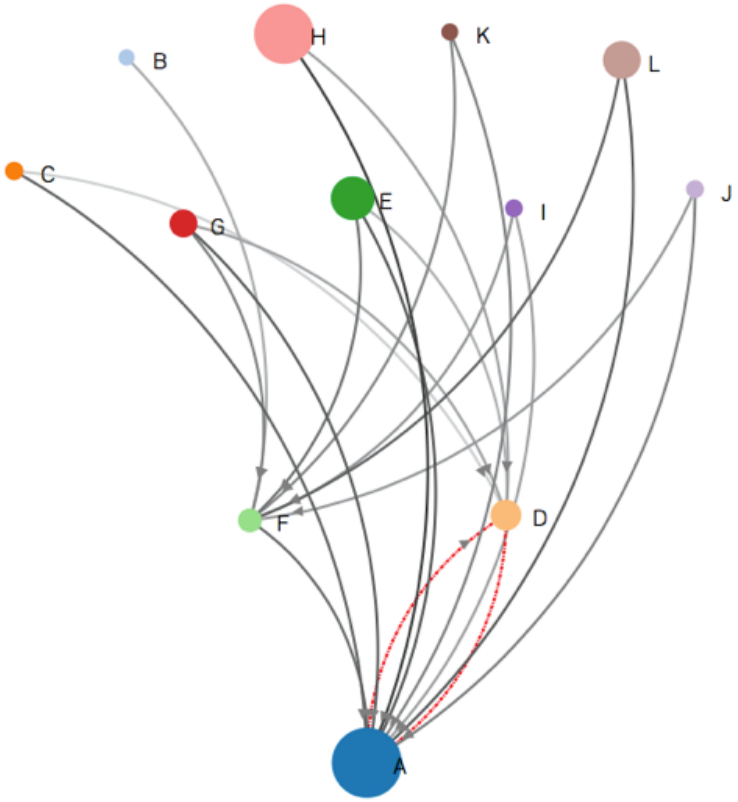


Fig. 7.2 Cyclic dependency

Cyclic dependencies between A and D as marked in red dashed lines

7.3 Case Study 2

7.3.1 Background

System B is a traffic management system. It contains a large quantity of information covering the fields of traffic, weather, and other conditions to administrate, manage and inspect. The development is in a transition state since the team is restructuring the system. Therefore, system B experiences a large number of changes, which is expected by the team.

Within system B, technologies used are Java, JavaScript, Html and CSS. Among them, Java is the major technology so that the visualization focuses on components developed in Java. We visualized the evolution from 2009 to 2016 which almost covered its whole history.

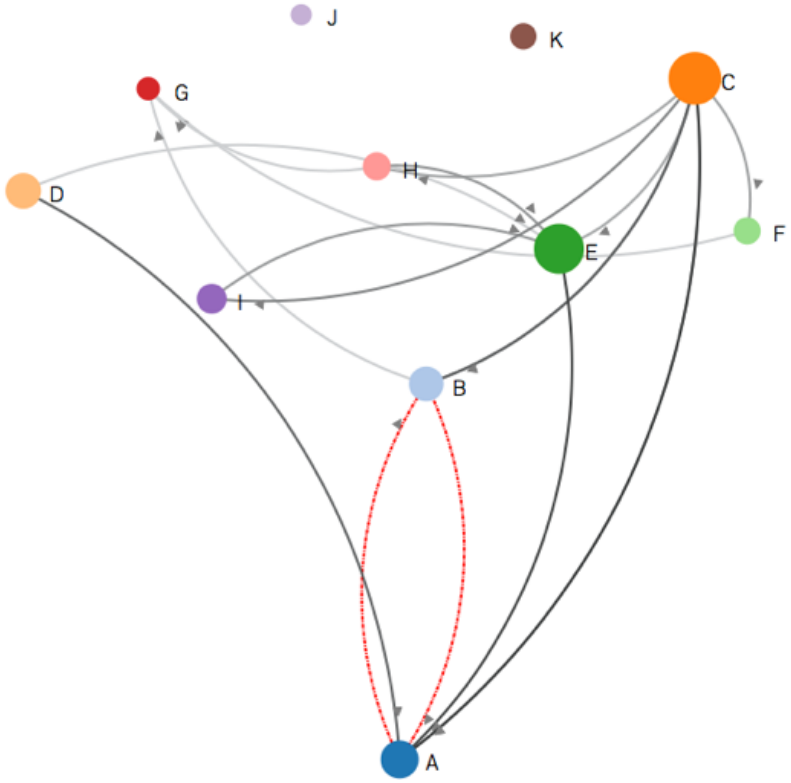


Fig. 7.3 Cyclic dependencies in system B

7.3.2 Interview

An interview was conducted with the development team and a consultant. During the interview, the visualized evolution history of system B was presented to them. The facts identified and conclusions drawn are listed below:

- At the beginning, system B is relatively small and only contains two components.
- Around 2015, system B started to grow with a bunch of new components coming out.
- During the most recent half year, the architecture becomes more clear. The development team stated that they were implementing microservices [47], which is the reason why a set of small components appeared. Due to this re-architecting, the dependencies among components became more clear and the component size distribution became better. However, the result is not reliable since the new components caused by microservices will bring hidden dependencies, for example, cross technology dependencies. These hidden dependencies can't be identified by our tool. As described in Chapter 5, our data is collected by a static code analysis tool SAT which can't detect cross-technology dependencies. Thus these hidden dependencies can't be presented, which makes the dependency looks more clear.
- There are several cyclic dependencies identified. Some of them were solved immediately. However, one of them hasn't been solved yet as shown in Figure 7.3.
- As shown in Figure 7.3, component A and E are heavily dependent, which indicates that they could be split into smaller components.

In this case, some of the conclusions we draw from the visualization are confirmed by the consultant and the development team's future plan:

- The consultant also noticed the cyclic dependency and recommended the developers not to solve the cyclic dependency now. Since they were implementing microservices, component A would be split later and then the cyclic dependency would be solved naturally.
- The development team confirmed that in their future plan, component A and E are going to be split.

The development team are not intending to reach a stable architecture at that point and changes are not alarming to the team. Thus the purpose of the prototype is not to alert them to new changes, instead, it helps them to evaluate the architectural impacts of their work. They

regard the dependency changes as an KPI for the develop, especially removed dependencies and decreased dependencies since these changes indicate that they made the right decisions. With respected to identifying architectural violations, they said “it’s hard to see the direct effects now”, they’d like to have the visualization prototype as a daily monitor tool since it should be used on a regular basis to identify problems. In addition, this case shows that through the prototype, we can reach similar conclusions with consultants. In this way, it also confirms the usefulness of the prototype.

7.4 Case Study 3

Unlike the first two cases, case 3 starts with a specific goal which is to evaluate the scalability of a system’s architecture. Therefore, in this case study, we’d like to figure out how the visualization prototype can facilitate the scalability evaluation.

7.4.1 Background

System C is specialized in providing insurance and health service and developed in C#. System C is a median sized system with 12 components and 34 links at most. We visualized 20 snapshots in total from 2011 to 2016. Currently, system C is operating smoothly in one country. The future plan is to extend it into several other countries and inevitably, there will be a significant increase in the workloads and accordingly, a higher demand on response time and reliability.

Thus, through the architecture evolution visualization, we’d like to see how system C came into being and predict its future state.

7.4.2 Results

An interview was conducted with a software consultant who is going to evaluate system C’s scalability. Firstly, we presented the visualization of the first snapshot as shown in Figure 7.4. The following findings are drawn:

- Components G, H, F, E, A, J don’t have incoming dependencies and are placed on the top, which means they are not dependent by other components. We guess that they are applications and user interface.
- Components C and D are placed in the middle and we guess that they are logic control components

- Components B and I are heavily dependent by other components
- There is a cyclic dependency between component C and D. Except the cyclic dependency, system C's architecture is quite clear.
- Component E is isolated

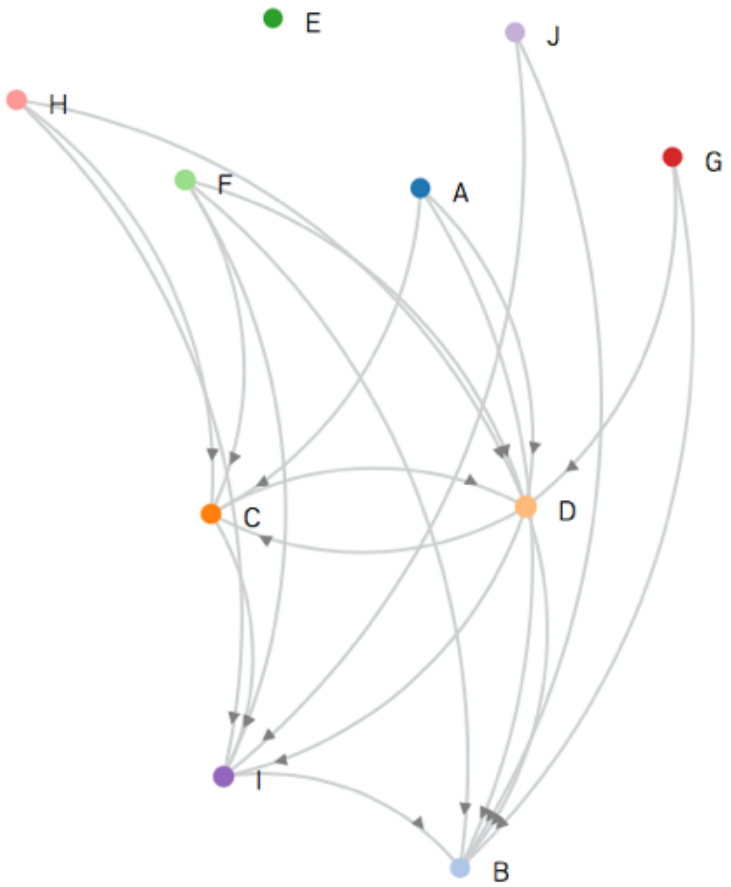
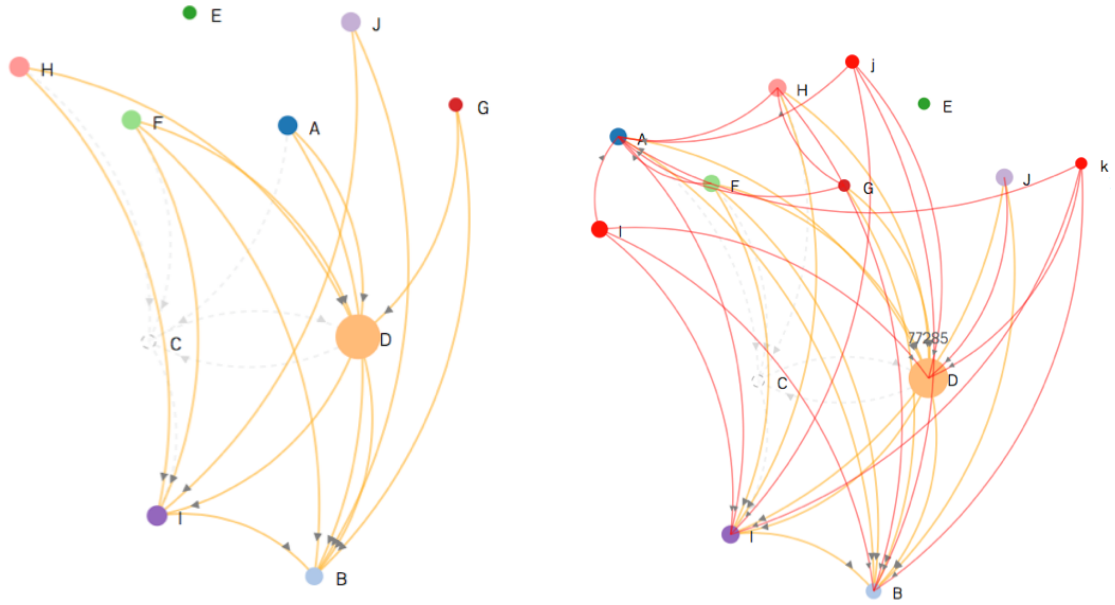


Fig. 7.4 The first snapshot visualization of system C

Snapshot Date: 2011-07-22

Then we compared the first snapshot with a later one as shown in Figure 7.5. To make it clear, Figure 7.5(a) ignores new components and their related dependencies and therefore only shows the changes of existing components. Figure 7.5(b) shows all the changes. Through the visualization, we draw the following conclusions:

- Almost all the components become bigger, especially component D which increased by 77285 lines of code



(a) Only shows the changes of existing components

(b) Shows all the changes

Fig. 7.5 Two snapshots comparison (2011-07-22 VS 2014-04-30)

- All the existing dependencies became stronger, which indicates the system is growing
- Component C and related dependencies are removed
- There are three new components: i, j and k (use lower case letters to present new components). All the new components depend on components D, B, I and A

In conclusion, the architecture is relatively stable during this period from 2011 to 2014. Newly added components also follows the same rules as existing components, for example, they all depend on component D, B and I which are heavily dependent by other existing components. System C has been enlarged several times in terms of both size and the number of dependency during the three years.

After that, we presented the evolution visualization. The last snapshot on 2nd May of 2016 is shown in Figure 7.6. Based on the evolution animation, it was obvious that component D became extremely large. At the same time, the color of the links directed to component D, B and I are getting darker, which indicates that these three components are more dependent on by other components. To take the analysis one step further, we check the percentage of the percentage of the four types of dependency (Section 6.1.2) of the biggest component D as shown in Figure 7.7, the percentage of throughput dependencies almost doubles. The consultant said it also indicated that component D becomes troublesome.

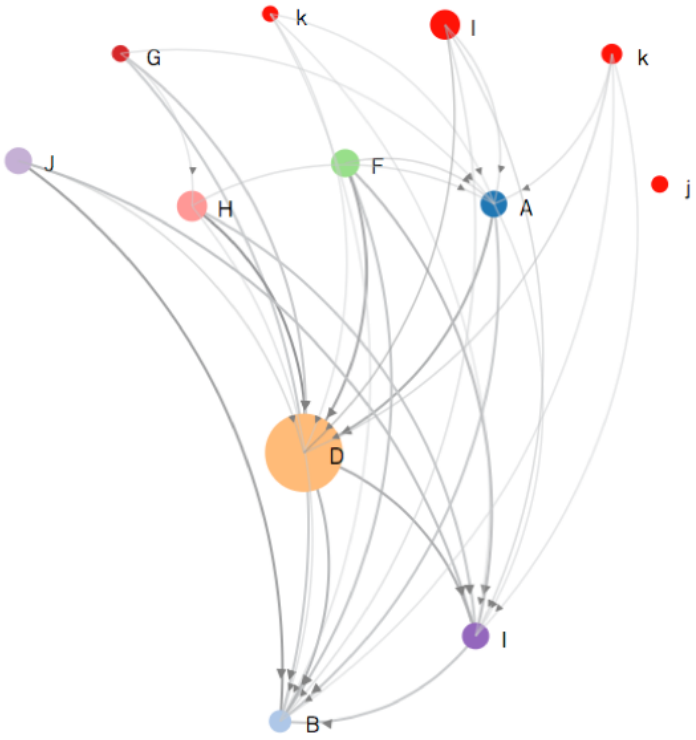


Fig. 7.6 The last snapshot of system C

snapshot date: 2016-05-02

Fig. 7.7 Component D: the percentage of the four types of dependency

On the left: the first snapshot. On the right: the last snapshot

In general, based on the visualization, we think in order to satisfy international usage, component D should be split into several smaller components, otherwise, the size of component D will grow too large. Component B and I are getting more dependent but without explicit size changes. Without proper adjustments, the maintenance expenses of these two components will be extremely high.

7.5 Conclusion

In this chapter, we implemented prototype validation through three case studies. The first two case studies aim at collecting stakeholder feedback towards the prototype with regard to its usefulness and applicability. The third case study starts with a specific problem for the purpose of assisting a consultant with architecture evaluation.

In general, the prototype is regarded as useful for stakeholders' work. The first case demonstrates that the prototype can be used as a communication tool and a monitor tool, which not only helps to present and communicate the existing work but also provides new insights for the development. The second case demonstrates that the prototype enables the development team to keep track of their work and evaluate the impacts at architecture level. The last case provides insights into applying the prototype to architecture evaluation consisting of current state analysis and future state predication. Furthermore, some participants think it's creative to combine architecture visualization with metrics.

Chapter 8

Conclusions

In this thesis, we focused on the research on implemented architecture evolution visualization of software systems. We followed the research framework (Section 1.3) covering "Why" ("Who"), "What" and "How" aspects of the visualization. To be more specific, four research questions are defined to guide the research (Section 1.2). This section will briefly go through all the questions to provide answers.

8.1 Answers to Research Questions

Question 1: *What are the main features of the implemented software architecture that matter during software development and maintenance?*

Through literature review (Section 3.3) and stakeholders' interviews (Section 4.1.1), we identified a set of architecture related features listed below in Table 8.1. Here we combined the results in Table 3.1 and Table 4.1.

Question 2: *Among existing visualization techniques, what are the best candidates for the features which is required by different stakeholders?*

In chapter 3, a set of existing visualization techniques is introduced with the emphasis on the features visualized and the way to visualize them. Then we conducted three interviews with stakeholders to identify the features stakeholders expected. By matching the features required by stakeholders and the features visualized by existing techniques, we defined the best candidates for the features required by stakeholders, the results can be found in Table 8.2 which is a copy of Table 4.1.

Question 3: *What is involved with implementing a visualization technique, which fits multiple stakeholders' needs of architecture evolution?*

Table 8.1 Architecture features

Metrics
Hierarchy of the system
The relationships between elements (components, modules, units)
Combination of hierarchical organization and relationship information
Hierarchical organization changes / Structural changes
Relationships (inheritance / call graphs / control-flow graphs) evolution
Technology
Deployment
Component changes
call chains
Co-evolution
Global overviews

Table 8.2 Match existing visualization techniques with stakeholder's requirements

"_": no matching techniques, "\": not applicable

Stakeholders	Features	Existing techniques (static aspect)	Existing techniques (Evolution)
Developer	Current states and changes of call graph, dependencies	Dependency Structure Matrix, SHriMP	Gevol
	Organizational changes	Treemap, SHriMP	Inheritance comparison
	Technology	_	_
	Deployment	_	_
Consultant	Component changes	Treemap, SHriMP	_
	Dependency changes, call chains	Dependency Structure Matrix, SHriMP	Gevol
	Co-evolution	\	_
Managers of development	Global overviews	HEBs, SHriMP	_

Table 8.3 An overview of the prototype's functionality and design

Scenario	Information	Visual encoding
Visualize Single Snapshot	Components and dependencies among them	Node-Link diagram with point marks as items and connection marks as links, all marks are encoded by multiple channels (See chapter 5.1.2)
Compare Two Snapshots	Changes on components and dependencies	Changes indicated by channels such as color, size, etc
Evolution	Architecture evolution	Animated dependency graph

After matching requirements and existing techniques, we found not all of the stakeholders' needs can be fulfilled by existing techniques, new techniques are still in need. Thus we selected the requirements that all the stakeholders have in common and translated them into functions, the results can be found in Table 4.2. Furthermore, we described visualization design of each scenario and the results are listed in Table 5.3. When finishing functionality design and visualization design, we started to build a web-based visualization prototype which is described in Chapter 6. The last step is to validate the visualization prototype by case study. Table 8.3 (a copy of Table 6.1) presents the overview of the prototype's functionality and design.

Question 4: *What can be learned from the visualization prototype?*

The visualization validation was conducted through three case studies, during which, we collected feedback on the prototype's usefulness and applicability (Section 7.2 and 7.3). In addition, we also use it to solve real problems in industry (Section 7.4). Almost all the participants gave positive feedback on the visualization prototype. They think the prototype can be used as a communication and a monitor tool. As a communication tool, the prototype can present the development work to less technical stakeholders and help to communicate the work with technical stakeholders. For example, developers can use it to report their work to their managers in an attractive and understandable way. As a monitor tool, the prototype enables developers to identify architecture violations, track their daily work and evaluate the architectural impacts. With respect to consultants, the prototype can assist them to conduct architecture evaluation including both current state analysis and future state predication.

8.2 Discussion

This research is subjected to several limitations. Limitations with regards to data collection, visualization design and validation are identified and described below.

8.2.1 Data Collection Limitations

Data collection has direct influence on the results of this research, therefore, in this section, the limitations of data collection covering literature review, interviews and visualization dataset collection are described.

Literature Review Limitations

In this research, the study of existing techniques through literature review is the foundation of the prototype design. Based on the results, we identified architecture related features and then matched them with stakeholders' requirements. Due to time constraints, the literature review is mainly depending on several survey papers that providing overviews of visualization techniques, which means the study is not exhaustive and some related techniques may be missed. Nevertheless, it's believed that the literature study of existing techniques was broad enough to suit the purpose of this research.

Interview Limitations

The requirement analysis is conducted through three interviews. In terms of identifying stakeholders' needs, the more diverse stakeholders we interview, the less biased the results are. In our case, we interviewed three stakeholders from SIG, which might lead to biased results due to similar backgrounds. The interviewees consist of a developer, a manager of development and software consultants, which doesn't cover all the roles during software development and maintenance. Nevertheless, the main goal of this research is to apply visualization techniques to software development and maintenance instead of identifying stakeholders' needs towards visualization and it's believed the input from the interviews is sufficient to meet the goal.

Dataset Limitations

The dataset of the visualization in this research is based on a SIG's internal static code analysis tool (SAT). Thus the results of the visualization are restricted by SAT. SAT can't identify cross-technology dependencies. Therefore, the prototype only presents the dependencies

between components using the same technology. However, with further development of SAT, or the use of more advanced tools this problem can be solved.

8.2.2 Visualization Design Limitations

Design is always a set of trade-offs in an arduous struggle to implement the best solution, for example, the tradeoff between the amount of information and the comprehensability of the visualization. It's impossible to find a perfect design without any defects. In our case, the comprehensability of the visualization will decrease with the growth of system's size (the amount of components and dependencies). As mentioned in Chapter 5, we selected node-link diagram and the idea behind that is that the ratio of link to node should be smaller than four and nodes should be limited to dozens. Although a group of user interaction techniques were used to minimize the effects, the visualization's expressiveness is still not satisfactory when the amount of components and dependencies exceeds the required amount. Nevertheless, based on the component balance metric [8], the most common number of components of a system is around 8. The range of the number of components with good component balance score coincides with the range of the tool. For a system with hundreds of components, it's bad in terms of maintainability and the prototype is not designed for it. The prototype can work for the majority and the limitation is acceptable.

The prototype adopts animation as its approach to present the evolution. This approach reduces cognitive gaps and increases cohesion between snapshots. However, this approach has an obvious defect, which is that only one snapshot is presented each time and it requires users to remember previous snapshots and thus increases cognitive loads. In order to minimize the influence, the prototype supports two snapshot comparison as a compensation.

8.2.3 Validation Limitations

Another limitation of this research is related to the validation. We implemented prototype validation through three case studies. The first two case studies are intended to validate the usefulness and applicability of the visualization prototype. The third case study aimed at solving a real problem with the help of the prototype. Due to limited time and resources, only three case studies were conducted. It would be better if the prototype could be applied to a large number of different situations. In this way, more validated conclusions could be drawn and the usage of the prototype could be further explored. In addition, the participants are mainly developers and software consultants, which is insufficient to validate the usefulness for all stakeholders.

8.3 Future Work

We envision two areas of future work, which covers requirement analysis and prototype design.

8.3.1 Improving Requirement Analysis

Visualization techniques are widely used in software engineering area to assist multiple stakeholders. A visualization is more of a tool to help people think than just a beautiful picture. Currently, there are a great deal of visualization techniques, but only a small portion of them have been put into practice. One of the reasons might be that the visualizations can't solve stakeholder's pain points. Therefore, developing a visualization that is practical and functional is beneficial. It requires that visualizations should meet stakeholders' requirements in a user friendly way. In this research, only three stakeholders were interviewed, which is insufficient to provide an overview of all the stakeholder's needs of visualizations. One could conduct a research to collect the needs of more stakeholders involved in software development projects. The results can be regarded as the groundwork for future visualization techniques.

8.3.2 Improving Prototype Design

Functionality

With regards to functionality, the prototype provides the visualization of comparison of two snapshots and evolution of multiple snapshots of implemented architecture. One way to improve it is to add comparison between systems' implemented architecture and designed architecture. In this way, architecture erosion can be visualized and measured. In addition, the differences between the system's implemented architecture and designed architecture have high chance to be architecture violations, so that the prototype makes it easier to identify architecture violations.

Another functionality extension is to add separated views. Currently, the prototype presents all the information in a single view, for example, it shows component changes and dependency changes in a single visualization. It results in users' incomplete perception of the information displayed when there are a lot of changes on the graph. Thus, we think improving the prototype with separated views that only show one type of information can solve the problem.

Usability

In order to further promote the research prototype, its usability needs to be improved. One suggestion we got from case study candidates was to provide full screen mode. Another thing is to use deterministic layout. Since the starting horizontal position of components is given randomly, it's hard to compare with the previous visualization. Descriptions like "the component on the upper right corner" won't work after the visualization has been refreshed.

References

[1] Balzer, M., Deussen, O., and Lewerentz, C. (2005). Voronoi treemaps for the visualization of software metrics. In *Proceedings of the 2005 ACM symposium on Software visualization*, pages 165–172. ACM.

[2] Bass, L. (2007). *Software architecture in practice*. Pearson Education India.

[3] Bertin, J. (1983). Semiology of graphics: diagrams, networks, maps.

[4] Bladh, T., Carr, D. A., and Scholl, J. (2004). Extending tree-maps to three dimensions: A comparative study. In *Asia-Pacific Conference on Computer Human Interaction*, pages 50–59. Springer.

[5] Boehm, B., Bose, P., Horowitz, E., and Lee, M. J. (1995). Software requirements negotiation and renegotiation aids: A theory-w based spiral approach. In *Software Engineering, 1995. ICSE 1995. 17th International Conference on*, pages 243–243. IEEE.

[6] Bosch, J., Gentleman, M., Hofmeister, C., and Kuusela, J. (2013). *Software Architecture: System Design, Development and Maintenance*, volume 97. Springer.

[7] Bostock, M., Ogievetsky, V., and Heer, J. (2011). D^3 data-driven documents. *IEEE transactions on visualization and computer graphics*, 17(12):2301–2309.

[8] Bouwers, E. M. (2013). *Metric-based Evaluation of Implemented Software Architectures*. TU Delft, Delft University of Technology.

[9] Card, S. K., Mackinlay, J. D., and Shneiderman, B. (1999). *Readings in information visualization: using vision to think*. Morgan Kaufmann.

[10] Carpendale, S. and Ghanam, Y. (2008). A survey paper on software architecture visualization.

[11] Caserta, P. and Zendra, O. (2011). Visualization of the static aspects of software: A survey. *IEEE transactions on visualization and computer graphics*, 17(7):913–933.

[12] Collberg, C., Kobourov, S., Nagra, J., Pitts, J., and Wampler, K. (2003). A system for graph-based visualization of the evolution of software. In *Proceedings of the 2003 ACM symposium on Software visualization*, pages 77–ff. ACM.

[13] Dachselt, R. and Ebert, J. (2001). Collapsible cylindrical trees: a fast hierarchical navigation technique. In *Proceedings of the IEEE Symposium on Information Visualization (InfoVis 2001), San Diego*.

[14] Diehl, S. (2007). *Software visualization: visualizing the structure, behaviour, and evolution of software*. Springer Science & Business Media.

[15] Dix, A. and Ellis, G. (1998). Starting simple: adding value to static visualisation through simple interaction. In *Proceedings of the working conference on Advanced visual interfaces*, pages 124–134. ACM.

[16] Eick, S. G., Graves, T. L., Karr, A. F., Mockus, A., and Schuster, P. (2002). Visualizing software changes. *IEEE Transactions on Software Engineering*, 28(4):396–412.

[17] Gall, H., Jazayeri, M., and Riva, C. (1999). Visualizing software release histories: The use of color and third dimension. In *Software Maintenance, 1999.(ICSM'99) Proceedings. IEEE International Conference on*, pages 99–108. IEEE.

[18] Garlan, D. (2000). Software architecture: a roadmap. In *Proceedings of the Conference on the Future of Software Engineering*, pages 91–101. ACM.

[19] Garlan, D. and Shaw, M. (1993). An introduction to software architecture. *Advances in software engineering and knowledge engineering*, 1(3.4).

[20] Ghoniem, M., Fekete, J.-D., and Castagliola, P. (2004). A comparison of the readability of graphs using node-link and matrix-based representations. In *Information Visualization, 2004. INFOVIS 2004. IEEE Symposium on*, pages 17–24. Ieee.

[21] Gogolla, M., Radfelder, O., and Richters, M. (1999). Towards three-dimensional representation and animation of uml diagrams. In *International Conference on the Unified Modeling Language*, pages 489–502. Springer.

[22] Gutwenger, C., Jünger, M., Klein, K., Kupke, J., Leipert, S., and Mutzel, P. (2003). A new approach for visualizing uml class diagrams. In *Proceedings of the 2003 ACM symposium on Software visualization*, pages 179–188. ACM.

[23] Halik, Ł. (2012). The analysis of visual variables for use in the cartographic design of point symbols for mobile augmented reality applications. *Geodesy and Cartography*, 61(1):19–30.

[24] Hindle, A., Jiang, Z. M., Koleilat, W., Godfrey, M. W., and Holt, R. C. (2007). Yarn: Animating software evolution. In *2007 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis*, pages 129–136. IEEE.

[25] Holten, D. and Van Wijk, J. J. (2008). Visual comparison of hierarchically organized data. In *Computer Graphics Forum*, volume 27, pages 759–766. Wiley Online Library.

[26] Holten, D. H. R. (2009). Visualization of graphs and trees for software analysis.

[27] Johnson, B. and Shneiderman, B. (1991). Tree-maps: A space-filling approach to the visualization of hierarchical information structures. In *Visualization, 1991. Visualization'91, Proceedings., IEEE Conference on*, pages 284–291. IEEE.

[28] Kaufmann, M. and Wagner, D. (2003). *Drawing graphs: methods and models*, volume 2025. Springer.

[29] Kazman, R., Bass, L., Webb, M., and Abowd, G. (1994). Saam: A method for analyzing the properties of software architectures. In *Proceedings of the 16th international conference on Software engineering*, pages 81–90. IEEE Computer Society Press.

[30] Khan, T., Barthel, H., Ebert, A., and Liggesmeyer, P. (2012). Visualization and evolution of software architectures. In *OASIcs-OpenAccess Series in Informatics*, volume 27. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[31] Kogut, P. and Clements, P. (1994). The software architecture renaissance. *Crosstalk-The Journal of Defense Software Engineering*, 7(11):1–5.

[32] Krygier, J. and Wood, D. (2011). *Making maps: a visual guide to map design for GIS*. Guilford Press.

[33] Lehman, M. M. and Belady, L. A. (1985). *Program evolution: processes of software change*. Academic Press Professional, Inc.

[34] Lehman, M. M., Ramil, J. F., Wernick, P. D., Perry, D. E., and Turski, W. M. (1997). Metrics and laws of software evolution-the nineties view. In *Software Metrics Symposium, 1997. Proceedings., Fourth International*, pages 20–32. IEEE.

[35] Mackinlay, J. D. (2000). Opportunities for information visualization. *IEEE Computer Graphics and Applications*, 20(1):22–23.

[36] Mens, T., Wermelinger, M., Ducasse, S., Demeyer, S., Hirschfeld, R., and Jazayeri, M. (2005). Challenges in software evolution. In *Eighth International Workshop on Principles of Software Evolution (IWPSE'05)*, pages 13–22. IEEE.

[37] Morrison, J. L. (1974). A theoretical framework for cartographic generalization with the emphasis on the process of symbolization. *International Yearbook of Cartography*, 14(1974):115–27.

[38] Munzner, T. (2014). *Visualization Analysis and Design*. CRC Press.

[39] Parnas, D. L. (1994). Software aging. In *Proceedings of the 16th international conference on Software engineering*, pages 279–287. IEEE Computer Society Press.

[40] Randelshofer, W. (2011). Visualization of large tree structures.

[41] Rohrer, R. M. and Swing, E. (1997). Web-based information visualization. *IEEE Computer Graphics and Applications*, 17(4):52–59.

[42] Sangal, N., Jordan, E., Sinha, V., and Jackson, D. (2005). Using dependency models to manage complex software architecture. In *ACM Sigplan Notices*, volume 40, pages 167–176. ACM.

[43] Storey, M.-A., Best, C., and Michand, J. (2001). Shrimp views: An interactive environment for exploring java programs. In *Program Comprehension, 2001. IWPC 2001. Proceedings. 9th International Workshop on*, pages 111–112. IEEE.

[44] Storey, M.-A. D., Čubranić, D., and German, D. M. (2005). On the use of visualization to support awareness of human activities in software development: a survey and a framework. In *Proceedings of the 2005 ACM symposium on Software visualization*, pages 193–202. ACM.

[45] Tamassia, R. (2013). *Handbook of graph drawing and visualization*. CRC press.

[46] Thomas, J. J. (2005). *Illuminating the path:[the research and development agenda for visual analytics]*. IEEE Computer Society.

[47] Thönes, J. (2015). Microservices. *IEEE Software*, 32(1):116–116.

[48] Verlet, L. (1967). Computer" experiments" on classical fluids. i. thermodynamical properties of lennard-jones molecules. *Physical review*, 159(1):98.

[49] Von Mayrhauser, A. and Vans, A. M. (1995). Program comprehension during software maintenance and evolution. *Computer*, 28(8):44–55.

[50] Yi, J. S., ah Kang, Y., Stasko, J., and Jacko, J. (2007). Toward a deeper understanding of the role of interaction in information visualization. *IEEE transactions on visualization and computer graphics*, 13(6):1224–1231.

Appendix A

Invitation Letter

Software is the DNA of our information society, thus people must make sure the smooth, predictable, accurate operation of software. However, it's not an easy task since software is intangible and needs to evolve all the time. Evolving software quickly becomes complex which results in a boost in the time and effort needed to understand, analyse and maintain it.

Visualization can help a lot to easy the task by providing visible measures to examine software components. This research is to develop a software architecture evolution visualization prototype, which focuses on the implemented architecture evolution at the component level. It contains the evolution information of both components and dependencies. It tells you where your system evolves and captures the major changes. It also aims at helping the development team to find potential problems in the system. Furthermore, it supports interactive navigation and animation, which intends to deliver message in an effective way. This is opposed to existing approaches, where the evolution is typically shown using static visualizations.

We sincerely invite you to participate in the validation phase of this research, consisting of two steps:

1. We apply the architecture evolution visualization prototype on your system and show you the results.

2. Research review session. It will take around 30 minutes. This interview consists mainly the following parts:

   (a) To what extent, the prototype helps you to understand the system?

   (b) Can the prototype help to find potential problems?

   (c) How can the prototype help you in your daily work?

   (d) General comments and improvements

All the study results will remain anonymous.

We are looking forward to your participation!

Shenshen Fan (Software Improvement Group & Leiden University)