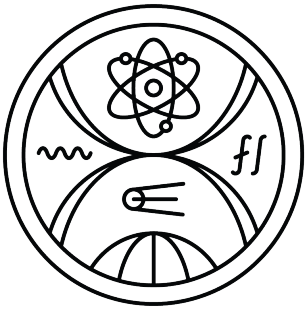
COMENIUS UNIVERSITY IN BRATISLAVA
FACULTY OF MATHEMATICS PHYSICS AND INFORMATICS

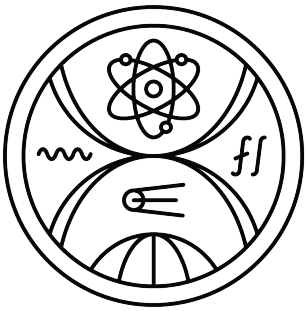


IDENTIFICATION AND VISUALIZATION OF SOFTWARE ARCHITECTURES

Master thesis

2024 Bc. Marek Dinka

COMENIUS UNIVERSITY IN BRATISLAVA
FACULTY OF MATHEMATICS PHYSICS AND INFORMATICS



IDENTIFICATION AND VISUALIZATION OF SOFTWARE ARCHITECTURES

Master thesis

Study program:	Applied informatics
Branch of study:	Applied informatics
Department:	Department of Applied Informatics
Supervisor:	doc. Ing. Ivan Polášek, PhD.

Bratislava, 2024 Bc. Marek Dinka

Comenius University Bratislava
Faculty of Mathematics, Physics and Informatics

THESIS ASSIGNMENT

Name and Surname:	Bc. Marek Dinka
Study programme:	Applied Computer Science (Single degree study, master II. deg., full time form)
Field of Study:	Computer Science
Type of Thesis:	Diploma Thesis
Language of Thesis:	English
Secondary language:	Slovak

Title: Identification and visualization of software architectures

Annotation: Documenting architecture is essential for understanding and reviewing software. Existing tools are usually capable of reverse engineering source code to basic UML diagrams
or identifying basic anti-patterns and idioms to identify errors or code smells to assist the QA process.
The main topic of this work is to explore options for automated or semi-automated identification, extraction and deduction on the software architectural level.

Aim: Design and create a prototype of the toolset capable of reverse engineering large and real software systems to classify and identify architecturally important component and their relations.
Propose methods and implement basic concept to use extracted information and derived relations. Document extracted information in the form of textual and visual architectural views.

Literature: Anquetil, N. et al. (2020). Modular Moose: A New Generation of Software Reverse Engineering Platform. In: Ben Sassi, S., Ducasse, S., Mili, H. (eds) Reuse in Emerging Software Engineering Practices. ICSR 2020. Lecture Notes in Computer Science(), vol 12541. Springer, Cham. https://doi.org/10.1007/978-3-030-64694-3_8

Holger M. Kienle, Hausi A. Müller:
Rigi—An environment for software reverse engineering, exploration, visualization, and redocumentation,
Science of Computer Programming, Volume 75, Issue 4, April 2010, Pages 247-263

Supervisor:	doc. Ing. Ivan Polášek, PhD.
Department:	FMFI.KAI - Department of Applied Informatics
Head of department:	doc. RNDr. Tatiana Jajcayová, PhD.
Assigned:	21.11.2023
Approved:	05.12.2023

prof. RNDr. Roman Ďurikovič, PhD.
Guarantor of Study Programme

Comenius University Bratislava
Faculty of Mathematics, Physics and Informatics

.....
Student

.....
Supervisor

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Bc. Marek Dinka
Študijný program: aplikovaná informatika (Jednoodborové štúdium, magisterský II. st., denná forma)
Študijný odbor: informatika
Typ záverečnej práce: diplomová
Jazyk záverečnej práce: anglický
Sekundárny jazyk: slovenský

Názov: Identification and visualization of software architectures
Identifikácia a zobrazenie softvérových architektúr

Anotácia: Dokumentácia architektúry je nevyhnutná na pochopenie a preskúmanie softvéru. Existujúce nástroje sú zvyčajne schopné spätného inžinierstva zdrojového kódu na základné diagramy UML
alebo identifikovať základné antivzory a idiómy na identifikáciu chýb alebo kódových pachov, ktoré pomáhajú QA procesu.
Úlohou tejto práce je preskúmať možnosti automatizovanej alebo poloautomatizovanej identifikácie, extrakcie a dedukcie na úrovni architektúry softvéru.

Cieľ: Navrhnite a vytvorte prototyp sady nástrojov, schopných reverzného inžinierstva skutočných rozsiahlych softvérových systémov na klasifikáciu a identifikáciu architektonicky dôležitých komponentov a ich vzťahov.
Navrhnite potrebné metódy a implementujte základy koncepcie na využitie extrahovaných informácií a derivovanych vzťahov. Zdokumentujte získané informácie vo forme textových a vizuálnych architektonických náhľadov.

Literatúra: Anquetil, N. et al. (2020). Modular Moose: A New Generation of Software Reverse Engineering Platform. In: Ben Sassi, S., Ducasse, S., Mili, H. (eds) Reuse in Emerging Software Engineering Practices. ICSR 2020. Lecture Notes in Computer Science(), vol 12541. Springer, Cham. https://doi.org/10.1007/978-3-030-64694-3_8

Holger M. Kienle, Hausi A. Müller:
Rigi—An environment for software reverse engineering, exploration, visualization, and redocumentation,
Science of Computer Programming, Volume 75, Issue 4, April 2010, Pages 247-263

Vedúci: doc. Ing. Ivan Polášek, PhD.
Katedra: FMFI.KAI - Katedra aplikovanej informatiky
Vedúci katedry: doc. RNDr. Tatiana Jajcayová, PhD.

Dátum zadania: 21.11.2023

Dátum schválenia: 05.12.2023

prof. RNDr. Roman Ďurikovič, PhD.
garant študijného programu

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

.....
študent

.....
vedúci práce

I hereby declare that I have written this thesis by myself, only with help of referenced literature, under the careful supervision of my thesis advisor.

.....

Bratislava, 2024 Bc. Marek Dinka

Acknowledgement

Abstract

Keywords:

Abstrakt

Kľúčové slová:

Contents

List of Figures

List of Tables

Terminology

Terms

- **Level of abstraction**
  The conceptual size of software designers building blocks

Abbreviations

-

Introduction

TODO These days, there are many many different frameworks, libraries, languages as well as other .... Each of these tends to define their own rules, syntaxes, procedures and other limitations and restrictions. Thus if we want to visualize the architectures in which these components are used, we must first define a common abstraction <cite visualization book> by which we will be able to understand and define them, let us therefore start with basics ... module, module's interface, API

Unify format of titles -> i.e. decide between Good Title and Good title

Motivation

Chapter 1

Knowledge base

In this chapter, we will establish the knowledge base for the discipline in which this work is taking place. We will begin by looking at software architecture and the ways in which it influences software development. We will then continue with the main subject of this work, which is architecture recovery. Followed by a short dive into architecture visualization and finish with a introduction into the toolkit we will be improving, called Cinderella.

1.1 Software architecture

1.1.1 Definition

Software systems are abstract and intangible. They are not constrained by the properties of materials, nor are they governed by physical laws or by manufacturing processes. Because of the lack of physical constraints, they can quickly become extremely complex, difficult to understand and expensive to change [31].

The goal of software architecture is to lower the complexity, ease the understanding and lessen the expense of change in software systems by providing a blueprint to the system. A software system's architecture is the set of principal design decisions made about the system [32]. An Architectural Design decision is a notion which encompasses all architecturally relevant aspects of the system under development. These include [32]:

- Design decisions related to system's structure.
- Design decisions related to functional behaviour.
- Design decisions related to interaction.
- Design decisions related to the system's non-functional properties.

- Design decisions related to the system's implementation.

By providing such a blueprint for the system, we will lower its complexity, as all interested parties will now have the opportunity to observe said system from a higher level of abstraction and recognize otherwise obscured connections and relations. We will ease the burden of understanding by showing the initial idea behind this system together with its evolutions. Lastly we will lessen the expense of any potential changes required on the system by highlighting all the important points where these changes will need to take effect.

The form which components of this blueprint take varies, but a noteworthy part is often devoted to architectural styles and patterns. Architectural styles are designed to capture knowledge of effective designs for achieving specified goals within a particular application context. An architectural pattern is a named collection of architectural design decisions that are applicable to a recurring design problem, parametrized to account for different software development contexts in which that problem appears [32]. These both serve as the common knowledge base shared among all who come in contact with said blueprint and enable them to communicate on a higher level of abstraction.

1.1.2 Architectural styles

An important pillar of modern software architecture is the use of architectural styles in software architecture [7, 3]. An architectural style defines a family of related systems, typically by providing a domain-specific architectural design vocabulary together with constraints on how the parts may fit together [18]. This vocabulary is of great benefit to architects, who can use it to easily describe highly abstract concepts. Also to developers who often understand how a system described by this vocabulary should be implemented and can access countless resources related to this topic. And lastly to testers who can get an idea of how the implementation will look without inspecting the code, as well as many others. Another great benefit of styles is presented in their modularity, they define a set of systems, which may, if implemented correctly, be reused in multiple independent systems at little cost. Examples of architecture styles range from the very generic, such as client-server and pipes-filters [29], to very domain-specific, such as NASA's Mission Data Systems style [11] and the IEEE Distributed Simulation Standard [19].

1.2 Software architecture recovery

1.2.1 Motivation

One great struggle which often arises in big software systems, comes from their tendency to relentlessly evolve. This phenomenon is often refereed to as design drift - implementation of new and new requirements has changed the system so much, that its original architecture and design are almost impossible to recognize [23] - and is in concordance with Lehman and Belady's laws of software evolution [21].

The issue of design drift is exacerbated by the often non-existent or limited state of documentation. Its absence limits not only the understanding of system in its current state, but also of the design decisions taken during its development. Another problem arises from a high rate of turnover among information technology professionals, who, more often than not, carry away with them essential knowledge of the inner system's workings which can hardly be replicated even in a throughout documentation [22].

The ability to understand the underlying architecture of a software system is essential for all activities related to the system, these include:

- Implementation of new requirements. Most new or changed requirements require some changes within the system it-self. Even if the system we are changing is well tested and maintained, such changes are riddled with risks in unforeseen forms if their author does not understand the system's architecture and grasp the potential impacts of his actions.

- Reuse of a software system or its components. Often it is much easier, or even essential, to reuse an old system rather than develop a new one. This is often a monumental task, which requires a lot of effort and for it to be successfully an overview of where any potential problems may arise and where the two systems should be connected is paramount.

- Software system's maintenance. The effort and cost of a software system's maintenance often dominate the activities in a software system's lifecycle. Understanding and updating software system's architecture presents a critical facet of system's maintenance, as it often helps the system's maintainer to identify and resolve any problems within hours or days rather than weeks and months [13].

1.2.2 Definition

Architecture recovery represents the process of building architectural models from system's artifacts [13]. In this case, a system's artifact is a sparsely defined concept and represents any important piece of information which can be extracted from a system

to help in the recovery of its architecture. This can be a reference to class in code, an import of an external library, an architectural pattern used in the system, a specific behaviour observed during its execution, an issue reported by a user of the system or even an email between two of the system's developers [17].

Architecture recovery is thus composed of two steps, first comes the extracting of system's artifacts from source code, documentation, issue tracking system, communications and many others [17]. When enough artifacts have been extracted, we must focus on the second step, which consists of using these artifacts to produce estimations of a system's architecture.

Often a major role in the first step is assigned to code analysis. A source code is an inseparable part of every system and within it's lines is hidden the truest approximation of said system's architecture. Analysis of this kind can be divided into two groups [8, 36]

- Static analysis, in this case the analysis is focused on the written code itself. Various parsers and other more complex tools can be used to extract the system's class structure (in case of OOP systems), interactions between classes, technologies used by the system and many other architecturally important artifacts.

- Dynamic analysis, this option is focused on the execution of the system itself. The system's runtime behaviour is examined and artifacts are collected. Among these are low level details, such as function call stacks and values of variables at different points of execution, together with higher positioned data about the system's performance and memory usage.

One approach often investigated in the second step, is the attempt to produce a decomposition of the system [34, 1, 8, 33, 22]. Here, the goal is to divide a complex system into smaller more understandable parts, which may represent modules, components, or other architecturally relevant groupings. Such division, if correct, is highly valuable for the purpose of recovering architecture, as it enables us to divide the vast quantities of data into smaller more easily understood chunks [34].

Extraction of system's artifacts will not be explored further here, as this theme will be revisited in section 2.1, but we will expand upon system's decomposition, as this subject will be important to our later work.

1.2.3 System decomposition

The goal of decomposing system into smaller, architecturally relevant pieces, is one which has been investigated quite diligently [13]. Many different approaches have been suggested, implemented and tested [1, 22, 33]. Quite a few different categorisations of these approaches have also been invented and explored [17]. Among these, a distinctive example is presented by the Knowledge based vs a structure based approach.

A knowledge based approach is one build upon a understanding of what different pieces of source code, or even entities on a higher level of abstraction, do and how they behave. This knowledge is obtained from reverse engineering techniques, system's documentation as well as other sources. And is utilized during the decomposition of the system into parts which, together, are in alignment with some architectural style or idea [33].

In the structure based approach, one utilizes syntactic interactions (e.g. method calls or invocations) between entities and treats the system's decomposition as a optimisation problem, where the goal is to achieve the best values of cohesion, coupling and other factors relevant to the modular design of a system [33].

The last definition we need to establish, is the definition of architectural groupings into which the system will be decomposed. There are quite a few definitions to choose from and for the purposes of this work we will choose to define component and module.

A component is a software implementation that can be executed on a physical or logical device. A component implements one or more interfaces that are imposed upon it. This reflects that the component satisfies certain obligations. These contractual obligations ensure that independently developed components obey certain rules so that components interact (or can not interact) in predictable ways, and can be deployed into standard build-time and run-time environments [2].

D.L. Parnas, who developed the concept of modular programming, first defined module in [26] as A well-defined segmentation of the project effort, where each task forms a separate, distinct program module. At implementation time each module and its inputs and outputs are well-defined, there is no confusion in the intended interface with other system modules. At checkout time the integrity of the module is tested independently; there are few scheduling problems in synchronizing the completion of several tasks before checkout can begin. Finally, the system is maintained in modular fashion; system errors and deficiencies can be traced to specific system modules, thus limiting the scope of detailed error searching. Later in the same paper [26] he states that modularization of larger systems should be based on the principle of information hiding, defined by him as: Every module is characterized by its knowledge of a design decision which it hides from all others. Its interface or definition was chosen to reveal as little as possible about its inner workings. Later definitions concur with this idea: A module is a unit whose structural elements are powerfully connected among themselves and relatively weakly connected to elements in other units. Clearly there are degrees of connection, thus there are gradations of modularity. [4].

Thus, the difference between module and component seems obvious. Modules deal with code packaging and the dependencies among code, while components deal with implementing higher-level functionality and the dependencies among components. Components need their code dependencies managed, but they technically don't need a

module system to do it. [15]

1.3 Software architecture visualization

1.3.1 Definition

In [28] a systematic literature review has been made on the subject of architecture visualization techniques. In this study, the concepts of software visualization and software architecture visualization have been defined, two viewpoints have been identified, four types of visualization techniques have been recognized and ten categorizations of purposes have been characterized. We will now briefly touch upon each of these subjects and expand on those which relate to this work.

The practice of software visualization is defined as a visual representation of artifacts (such as requirements, design and program code) related to software and its development progress [10]. The practice of software architecture visualization is defined as a visual representation of architectural models and some or all of the architectural design decisions about these models [32].

Thus the difference between software visualization and software architecture visualization can clearly be defined by the level of abstraction on which each operates. Software visualization concerns it self with elements and behaviours closer to the actual implementation and codebase (e.g. within a component). Software architecture visualization works on a higher 'architectural' level, where behaviours and elements on the system's scale are considered (e.g. between components). An example of such consideration is the decomposition of a software system's architecture into layers, components or slices in a structural viewpoint, which is something we will also attempt. In this thesis, we will be working on the architectural level of abstraction and will thus be delving more deeply into the practice of software architecture visualization rather than that of software visualization.

Two distinct viewpoints of software architecture are also presented in [28]:

- First there is the structural viewpoint, which expresses software architecture with components and connectors and considers it as a high-level software structure of a system [5]. In other words this view focuses solely on the end products of a system and shows the system's architecture as viewed by someone who had no part in creating it.

- Second is the decisional viewpoint, its goal is to consider the decisions made during the process of creating and evolving architecture and to define software architecture as a set of these decisions together with their rationale [16].

The decisional viewpoint often requires more than just the implementation related artifacts (e.g. emails, artifacts from issue tracking system, ...) which we will target in our work and thus we will not investigate this viewpoint further. The structural viewpoint on the other hand, will be employed extensively as it collides with our goal of extracting architecture using a static analysis of the system.

Four types of visualization techniques are further declared in [28], an example of each can be seen in figure 1.1.

- The graph-based visualization technique (figure 1.1a) used nodes and links to represent the structural relationships between architecture elements. This technique is best used with big quantities of inputs as it puts more emphasis on the overall properties of a structure, rather than the type of each node.

- The notation-based visualization technique (figure 1.1b) uses nodes and links to represent the architectural elements themselves, unlike in the graph-based technique, the emphasis is given to the type of each node rather than the structure between them. This technique encompasses multiple modelling techniques among which are UML (unified modelling language) and archimate, both of which we will revisit later.

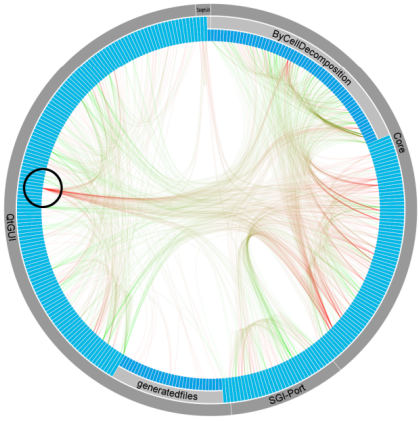
- The matrix-based visualization technique (figure 1.1c) often acts as a complementary representation of a graph. It can, for example, provide complementary information about the connections between graph's nodes when the graph is too large or dense for these to be shown.

- The metaphor-based visualization technique (figure 1.1d) uses familiar physical world contexts (e.g., cities) to visualize software architecture entities and their relationships. The use of methaphors makes the visualization process particularly intuitive and effective.

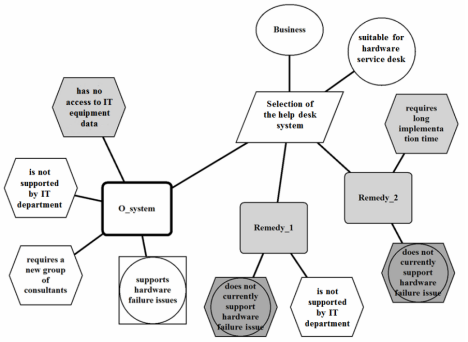
All of these techniques have their benefits and drawbacks as well as a type of information for which they are best suited. For our purposes we will focus mainly on the notation-based visualization technique, as it will be our goal to decompose the system into smaller parts and this technique will enable us to smoothly transform these parts into visual elements. We will also briefly touch upon the matrix-based visualization technique to visually compare greater amounts of data.

Lastly, in [28] there are also categorized ten purposes of using visualization techniques in software architecture. All of these categories can be seen in figure 1.2, we will expand only upon the ones relevant to this thesis.

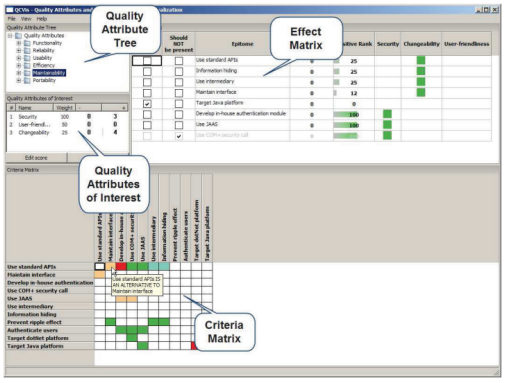
Category 2 Improve the understanding of static characteristics of architecture. Static characteristics of a software architecture do not change with execution of a system.



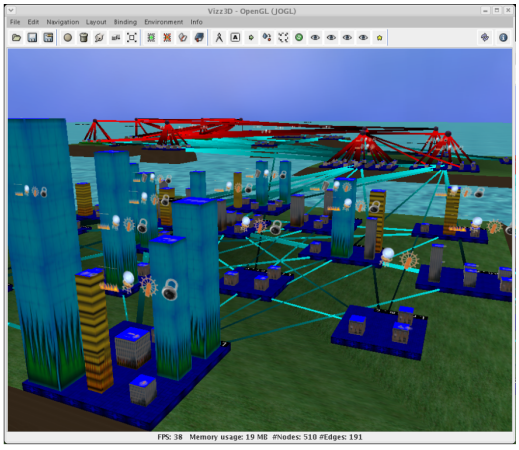
(a) Graph-based visualization [6].



(b) Notation-based visualization [35].



(c) Matrix-based visualization [9].



(d) Metaphor-based visualization [25].

Figure 1.1: Examples of visualization techniques defined in [28]

This means, that we are visualizing traits which can be deduced solely from static analysis of said system and can therefore also be deduced from system beyond the point of buildability or runnability. As such systems are among the main targets of the cinderella toolkit, this will be one of the main purposes behind our visualization.

Category 3 Improve search, navigation and exploration of architecture design. One of the goals behind the cinderella toolkit, is to extract and present as much architecturally relevant data as possible, this means that we will be often working with data on different levels of abstraction and in differing contexts. When working with such amounts of data, it is essential for it to be organized and navigable.

Category 7 Provide traceability between architecture entities and software artifacts. If we were to massively simplify architecture recovery into one main goal, the goal would be to take a software system, find its architecturally relevant artifacts and devise the system's architecture out of these artifacts. One of the products of

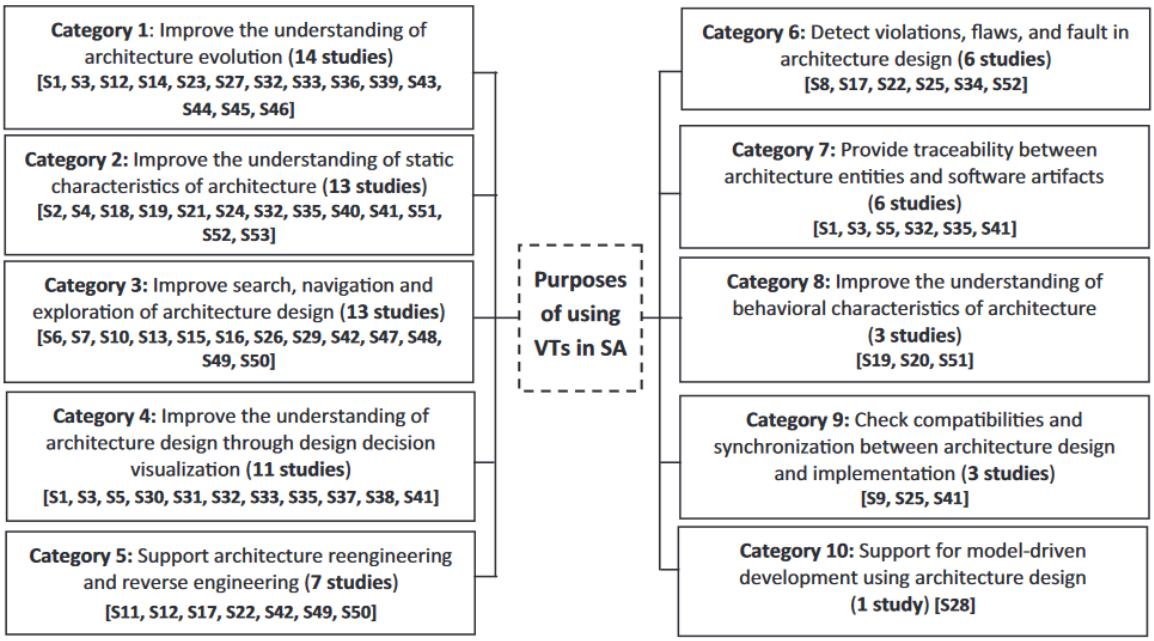


Figure 1.2: Categorization of the purposes for using visualization techniques in software architecture [28]

this process are the links between architecture's entities and software system's artifacts. The traceability is therefore quite easily obtained and the only hard part will be its visualization.

Category 8 Improve the understanding of behavioural characteristics of architecture. As with dynamic architecture analysis, the behavioural characteristics of an architecture are best observed during the runtime of a system, which is not something we will have available. Sometimes it is however also possible to find these characteristics in a static analysis (e.g. when a library whose behavioural characteristics are well known) and it would be a waste to ignore them. Thus this will not be a purpose with high priority but it will still belong among our purposes.

1.3.2 UML

We will now explore some of the technologies which we will be using for the purposes of software architecture visualization, starting with the Unified Modelling Language (UML). UML is a general-purpose visual modelling language, whose objective is to provide system architects, software engineers and software developers with tools for analysis, design and implementation of software-based systems as well as for modelling business and similar processes [24]. UML tries to fulfil this objective by using a wide range of diagrams, each with predefined scope and general purpose.

We have chosen UML for the purposes of our word due to its wide spread availability and usage, together with its great versatility. However, precisely due to this versatility,

we will be using only a small subset of the diagrams described in the UML standard, as many of the others are intended for purposes outside of the scope of this work. Thus, for our purposes, we have chosen to use the class and component diagrams.

- The main constituents of a static view of a system are classes, their relationships and various kinds of dependencies between them. A class diagram is a graphic presentation of a static view of a system, it is capable of representing and describing the system's classes, their contents and relations among them [27]. An example of such diagram can be viewed in figure 1.3.

- A component diagram shows the component based view of a system, it takes the software units out of which the system is constructed, puts them into a model and connects them based on their mutual dependencies [27]. An example of this diagram can be seen in figure 1.4

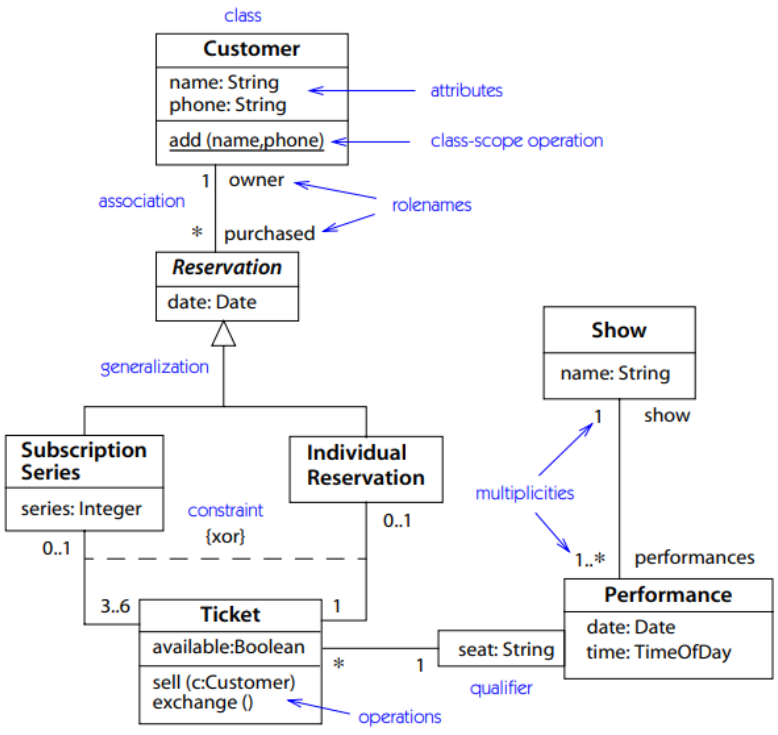


Figure 1.3: Example of a UML class diagram from [27]

Both of these diagrams are designed to show a static view of a system or its segment(s). The difference between them is in the level of abstraction on which they operate. Class diagram focuses on a level near source code and classes represented within it, while component diagram works on a much higher level with components and is often capable of showing the whole system in one diagram. Classes and components are both artifacts related to our work. Classes are already available for us

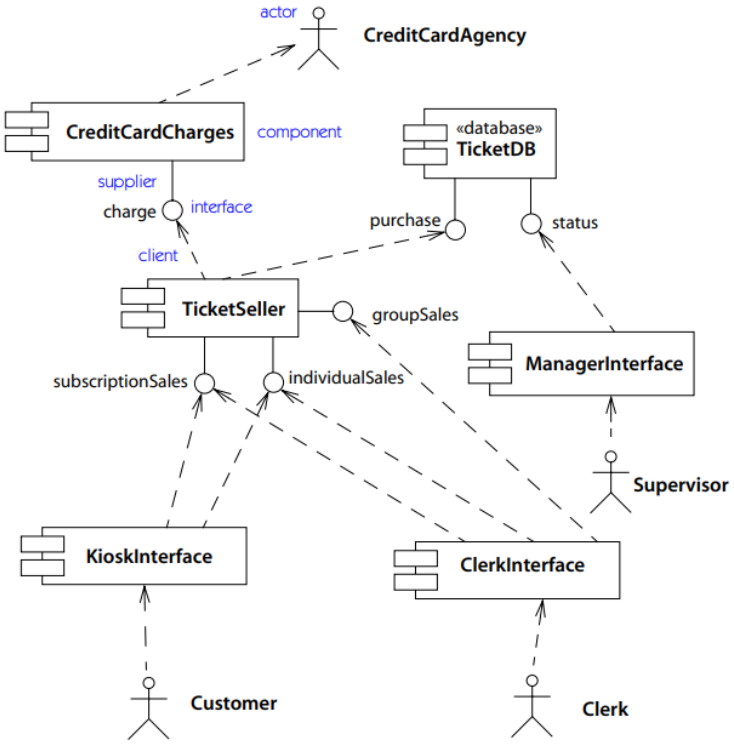


Figure 1.4: Example of a UML component diagram from [27]

from the cinderella toolkit and extracting components is one of the goals of this thesis. Thus, both of these diagrams are ideal for our software architecture visualization.

1.3.3 Archimate

Archimate is an enterprise architecture modelling language which supports the description, analysis and visualization of architecture within and across business domains in an unambiguous way [20]. As with UML, the scope of this language reaches way beyond what we are and will be able to recover. We will thus have to approach this language as we have approached UML and pick the parts we will be able to use.

The full framework of the Archimate modelling language can be seen in figure 1.5, it is composed out of six layers and four aspects. From these, the strategy and business layers are beyond what we aim to recover, application and technology layers will be the main layers used in this work, physical layer may also be touched upon, but if, then only briefly and the Implementation and Migration layer is also beyond what we aim to recover. From the aspects, recovering information which would enable us to use the motivation aspect would require a different type of analysis, so we will not use that one, we will however have a use for active structure and behavioural elements from the application layer, together with passive structure elements from both application and technology layers.

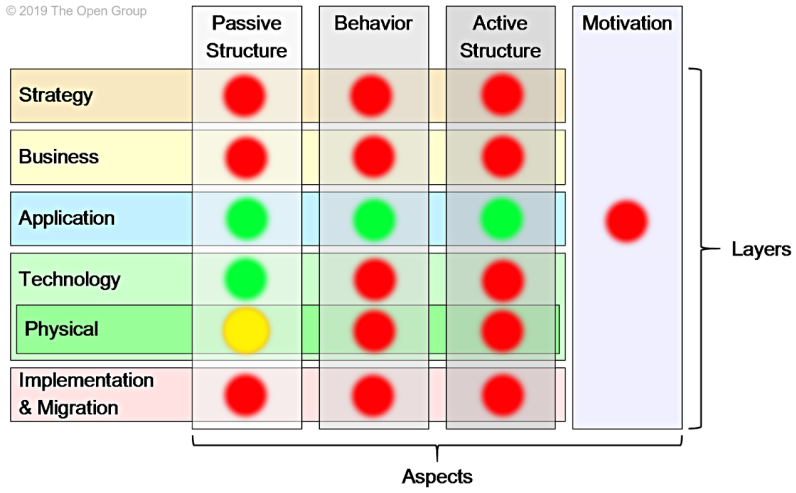


Figure 1.5: Full Archimate framework, from [14]. Green dots represent elements we will be using, yellow ones represent elements we may use and red dots are elements we will not use

- The application layer elements are typically used to model the application architecture that describes structure, behaviour and interaction of the application of the enterprise [14]. These are thus the elements which deal with the system it self, they can talk about its components, its interfaces, its processes... most of which represent the artifacts which we aim to recover. The application layer will be one of the main pillars of our visualization.

- The technology layer elements are typically used to model the technology architecture of the enterprise, describing the structure and behaviour of the technology infrastructure of the enterprise [14]. Technological artifacts can sometimes prove to be quite difficult to recover, as each technology likes to have its own specific syntax, rules, behaviour... An architecture recovery tool must therefore be aware of these specifics to be able to recover them. Fortunately, one of the goals of the cinderella toolkit is to try and recover just such artifacts and we will thus have a use for this layer as well.

- The physical layer elements are intended for modelling of the physical world [14]. Recovering artifacts related to the real world composition of a system, requires a very specific type of analysis which may only be applied to some systems. Such analysis will not be a part of this thesis, but we may encounter some artifacts which may hint at the state of real world infrastructure and may then have a use for this layer.

Chapter 2

Previous work

In this chapter we will explore a variety of previous works which contribute to the subjects of architecture recovery and architecture visualization.

2.1 The Cinderella toolkit

TODO -> revisit extraction of artifacts -> see section about architecture recovery Unlike many other tools, the cinderella toolkit concerns it self not only with source code, but focuses on many other artifacts

2.2 Architecture Recovery

2.2.1 Modular Moose

2.2.2 Architecture Recovery Using Cluster Ensembles

2.3 Architecture Visualization

Chapter 3

Research

Often, when we are trying to achieve some great goal, our path to this goal is littered with small steps and tasks which must first be fulfilled if we are to reach this goal. So it often is with architecture recovery and so it will be with this work. For us to be able to automatically or semi-automatically extract some architecturally important piece of information from a big software repository, we must first be able to grasp precisely what this piece of information is, how it should be and how it can be represented in a repository, out of what parts it is composed and how these can in turn also be detected. The following sections will describe my attempts at applying this process to various software repositories. We will talk about N different methods, which I have modified or created to assist this goal. Each method will be composed of the initial idea behind said method, the implementation of this idea and conclusions about its usage.

TODO -> talk about methods -> always mention one method, tell about it as follows:
idea behind the method
implementation of said method
conclusions about the benefits or fallbacks if this method -> also mention benefits for developer and architect

please note that the aim with the following methods was to explore the possibilities of that direction of research and not make a exhaustive research of said subject, thus their ideas often clash with imperfect understanding and their implementations are incomplete

3.1 Few Minor Methods

Here we will talk about a few methods more minor of nature and not deserving of a full section

3.1.1 Obtaining Interclass References

java_se_references

```
class A extends Reference1 implements Reference2 {
    Reference3 f() {
    }
};
```

Listing 1: add full input, say that it was used for testing references.

3.1.2 Exporting of trees into archimate

export-tree2archi

3.1.3 Analysis of Java Jar Files

jar-manifes-json

3.1.4 Abstraction Level

draw some pictures/show statistics

3.2 Modular decomposition

3.2.1 Idea

3.2.2 Implementation

3.2.3 Conclusions

i.e. modules (orange will also be mentioned somewhere in here (depending on viability of genetic algorithms it may get its own section))

Two different approaches to modular decomposition -> optimization problem approach (trying to get the highest values of some metrics) and comprehension driven approach (trying to understand the goals of said software as much as possible and clustering based on that (pattern driven approach) Tzerpos)

it seems that there are two distinct approaches towards software modular decomposition (see above) ... both have ups and downs, both are valid. I have choosen to attempt to improve the second approach, as the tool I am working with gives me great amounts of different small facts, which may help me to achieve this goal -> nevermind

we are trying to combine both of these goals -> cohesion+coupling for the first and homogeneous factor + ? for the other one

Another interesting point related to the two approaches is that, the main goal behind us decomposing a software system will play a significant role in choosing which one is better for us, i.e. if we have a software project and want to refactor and improve it, then the optimisation approach is ideal as it will give us the perfect modular decomposition as defined in ... (the book where module is defined). If we however wish to do what I aim to do in this work and try to understand the original architecture and intent of some software project, then the comprehension approach seems more benefitial.

The trouble with treating the modular decomposition as a optimization problem is that while we may be able to get the best combination of cohesion, coupling, modularization quality and other aspects. This ideal solution may not be very reflective of the actual intended architecture of a software project, as these measures were not really the main concerns of its creators (related to https://onlinelibrary.wiley.com/doi/epdf/10.1002/smr.2408)

3.3 Design structure matrix

3.3.1 Idea

[12]

3.3.2 Implementation

3.3.3 Conclusions

3.4 Archimate API

3.4.1 Idea

3.4.2 Implementation

3.4.3 Conclusions

3.5 File diversity chart

3.5.1 Idea

3.5.2 Implementation

3.5.3 Conclusions

3.6 File descriptors as genomes

3.6.1 Idea

In bioinformatics, a sequence alignment is a way of arranging the sequences of DNA, RNA, or protein to identify regions of similarity that may be a consequence of functional, structural, or evolutionary relationships between the sequences. citation from wiki, find some better one and use it

From my not so great understanding of this subject consider changing this, the science surrounding genomes often concerns it self with their comparisons and with obtaining insights based on these comparisons and their similarity. For the purposes of architecture recovery, this is a very interesting concept. If we were able to describe each file in a software repository with some simple descriptors, whose purpose would be to talk about that file's behaviour and aspects, then we might be able to apply the techniques and methods from the field of genome science to come to some conclusions about the architecture, which these files represent. Alternatively we might be able to define a genome for some specific architectural concept (e.g. a database component) and have a simple way of looking for it in the code.

In fact, this is something which cinderella is already capable of producing. We have detectors, small and simple bash scripts, whose only purpose is to collect small pieces of information about each and every file and save them in a format which can be easily

accessed by cinderella's developers.

3.6.2 Implementation

The first task, which must be addressed is choosing what type of information we will be putting into our genome. Our goal here would be for the genome to be as specific as possible, but at the same time we are only really interested in architecture related pieces of information and even these we would like to keep at a reasonable level, so that less important facts do not overwhelm the more important ones and thus mess with our calculations. With this in mind, when choosing the descriptors for each file, we will look at general detectors, important keywords, technologies and languages used in this file.

The next task, will be to choose the format of our generated genome. The FASTA format <citation or definition>, which we will be using for this purpose, states that each letter of its sequence represents a code for nucleotide or a amino acid <citation>, but since there are only 26 letters available in the alphabet, it is not viable for us to use such codes for our descriptors. It would also not be very beneficial to try to encode our descriptors using 2 or 3 letters, as the only result it would bring, would be to complicate the implementation. The correct solution thus seems to be, to use the descriptors as they are (without special characters).

We now have a set of descriptors, each of which is in a format acceptable by FASTA, and we have to decide how to join them into a single genome. There is a variety of options for how this could be done, but in this case the simplest option seems to also be the best. We will simply join these descriptors in a predefined order into a string and pronounce it to be a genome representing a file. For example a file from the jasperreports <Citation> project, called `CalendarUnit.java` would be described by the relevant descriptors in listing 2

From this, we would next generate the genome in listing 3 <Citation or definition of FASTA> also mention that it is in FASTA.

Thus generated genomes will then be sent to a tool called Clustal omega. Clustal omega is the latest instalment of the Clustal family of programs and can be used for performing fast and accurate multiple sequence alignments (MSAs) of potentially large numbers of protein or DNA/RNA sequences [30]. Or in other words, this is a tool which is widely used in the world of bioinformatics for its reliability as well as other great attributes. It can create alignments of genome sequences, clusters of genomes and a phylogenetic tree, all out of the genomes we put into it. The obvious course of action therefore, is to do just that, put all our genomes into it, and see what comes out.

First we will look at the clusters generated from these genomes, as this is a theme

```
{
  "file": "jasperreports/src/net/sf/jasperreports/types/date/CalendarUnit.java",
  "detectors": [
    "_ANY",
    "git_tracked",
    "__jasper",
    "contains_copyright",
    "jasper_api_jasperreports",
    "java_type_enum_public"
  ],
  "KEYWORDS": [
    "Source Control Tracked File",
    "Framework/Product",
    "Documentation",
    "Ownership",
    "API/Technology usage",
    "Programming Language"
  ],
  "TECHS_FMWKS_uniq": [
    "JasperReports"
  ],
  "LANGUAGES": [
    "Java"
  ]
}
```


Listing 2: Relevant descriptors for a file called `CalendarUnit.java` in json format.


```
>jasperreports/src/net/sf/jasperreports/types/date/CalendarUnit.java
ANYGITTRACKEDJASPERCONTAINSCOPYRIGHTJASPERAPIJASPERREPORTSJAVATYPEENUMPUBLIC
```


Listing 3: genome created from relevant descriptors, in FASTA format.

with which we have already some experience as well as results to compare these to. If we are to use these clusters, we must first understand what precisely they represent and for that we must figure out what Clustal omega has done to generate them. From our point of view, it has taken all the files we have provided it with and divided them into clusters based on the string similarity of their genomes. This might sound like a quite simple process, but the great advantage and the reason why we are using Clustal omega, is that it not only compares strings and parts of strings, but it also takes subsets of each string and tries to find them new places, together with many other comparison techniques, all with great efficiency, which is something we could not easily replicate.

One more aspect which must be considered is that in genomes, we are using the whole names of detectors and other attributes generated by Cinderella. Thanks to this a file's genome which contains the detector `java_se_version_8plus` will still be

a strong match with genome that contains the detector `java_ee_version_EE9plus`, despite the fact that from Cinderella's point of view these are two almost unrelated detectors.

Thus, what each of these clusters represents is a group of files which have been described by Cinderella in a similar manner. What this is to us, is a link between files and any detected components, or other architecturally relevant elements. The theory being, that if most files in a cluster share an attribute, or a group of attributes, which are related to, or straight up describe a component (or any other architecturally relevant element), then these attributes will be the components artifacts within the repository it self. A question may yet again arise, as to why do we even need these clusters, when we can group the files described by these attributes directly. The answer lays with the amount of attributes for each file. The average count of attributes for a file in the Jenkins project cite is 13 for 5541 out of 12485 files, with some files reaching all the way up to 53 attributes. With the direct grouping, we would only be able to account for one, maybe two of these attributes at a time, while with our approach, we can account for all 53 at the same time.

Now, if we are to actually use these links, we must first find a way of visualizing components and then incorporate said links to these components. For these exploratory purposes, the most optimal way need not be sought right away, we will thus take advantage of our previous experience with Archimate and use that. We take a list of all potential attributes (around 500 at the time of writing) and map all those, which may hint at a component or any other architecturally relevant piece of information to an Archimate element. We will then generate an Archimate view with all such elements found in a project (automatically, thanks to one of our previous methods). Next we will take each cluster, find those of its attributes which are present in majority of its files (for now, let's say $> 80\%$ of cluster's files) and use this information to link said cluster to the elements in the general view. An example of this view can be seen in figure 3.1

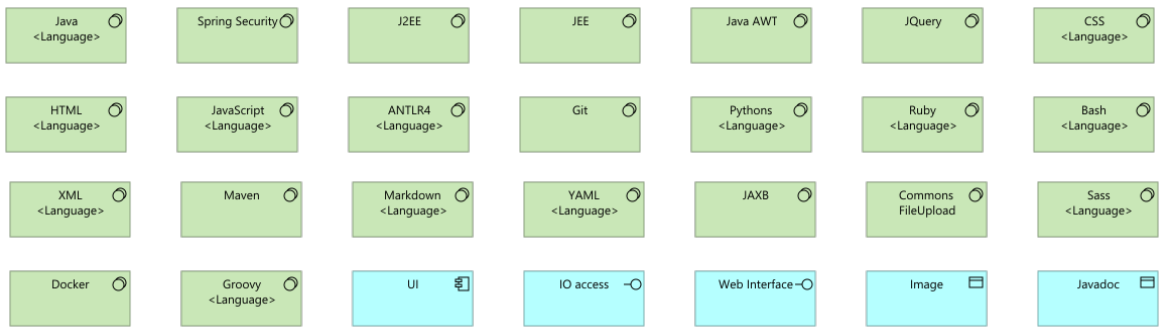


Figure 3.1: General view of elements from attributes detected in the Jenkins project cite

The problem with this approach, which soon becomes obvious, is the variability of all these attributes. Some speak of quite general concepts, like language and file type (e.g. the `java_type_` detector, which can detect java code), some tell us of concepts unrelated to our efforts (e.g. the `java_se_references`, which tells us that a file has references for other classes) and some, which often have a rather sparse occurrence, tell us of the concepts which interest us (e.g. the `spring_security_api` detector, whose responsibility is to find occurrences of the spring security technology).

This issue is best illustrated on an example, in figure 3.2 we can see a view generated from the same attributes as figure 3.1, but this time only elements with at least 80% presence in at least one cluster are generated.

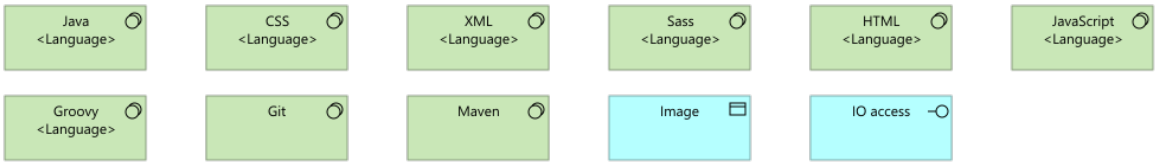


Figure 3.2: View of elements from clustered attributes in the Jenkins project cite

Another example can be seen in in listing 4, where can be observed detectors, which were activated on a file in the Jenkins project cite, next to the number of files on which they were activated. As we can see, some of the most occurring detectors (e.g. `git_tracked`, `_ANY`, `contains_copyright`, `java_ast`...) are just blurring the generated genomes by adding irrelevant (from the point of view of this method, not overall) data, which is then used for clustering and hides the relevant and essential pieces of information.

```
12483 git_tracked
12483 _ANY
 7597 properties_file
 7583 properties_file_l12n
 7468 contains_copyright
 1999 xml_content
 1762 java_ast
 1421 html_contains_markup
 1315 java_type_class_public
 1102 __TEST
 1068 xml_root_namespace_
 1000 html_html
  926 xml_root_name_div
...
```

Listing 4: Occurrences of detectors in the Jenkins project cite.

join this to the previous text The problem which must be resolved in this approach,

is the filtering of attributes. Some attributes (e.g language related ones) occur far more frequently than others and may hide other, more important ones. This then causes us to ignore these hidden ones, as we only consider the attributes, which have a majority presence in a cluster. An example can be seen in figure 3.3.

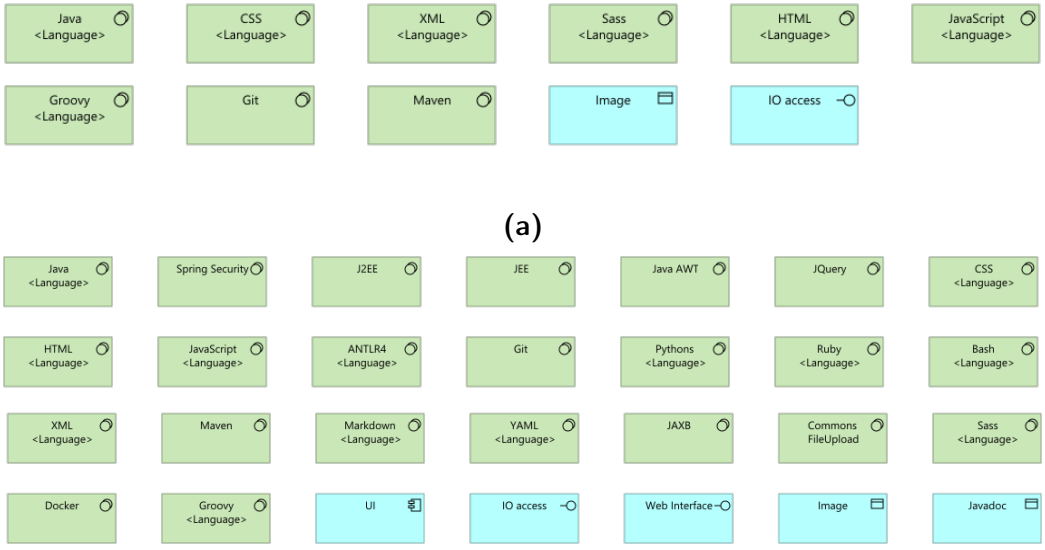


Figure 3.3: Archimate comparison of elements generated based on attributes extracted from the Jenkins project cite.(a) Elements with $\geq 80\%$ presence, (b) Elements from all attributes

Another example of this behaviour is shown in listing 5, where we can see unfiltered occurrences of detectors in the Jenkins cite project.

```
12483 git_tracked
12483 _ANY
 7597 properties_file
 7583 properties_file_l12n
 7468 contains_copyright
 1999 xml_content
 1762 java_ast
 1421 html_contains_markup
 1315 java_type_class_public
 1102 __TEST
 1068 xml_root_namespace_
 1000 html_html
  926 xml_root_name_div
...
```

Listing 5: Unfiltered occurrences of detectors in the Jenkins project cite.

If we were to add a filter for noisy detectors, a filter for general detectors and a

filter to remove any language detectors, the result would not be perfect, but it would be much better as can be seen in listing 6

```
   7597 properties_file
   1762 java_ast
    351 java_se_version_8plus
    351 java_lambda_expression
    329 java_Deprecated
    217 java_beans_methods
    191 java_ee_version_EE9plus
    141 image_gif
    134 java_synchronized
    123 java_class_Exception
    122 image_png
    114 spring_security_api
     72 java_lang_reflect
...
```

Listing 6: Filtered occurrences of detectors in the Jenkins project cite.

__TEST we want to keep, things like java_Deprecated and java_synchronized are not much relevant -> general model will be composed of all available attributes, there will be very specific clusterings -> each will generate some elements -> composed of relation

The information about languages however is one we would still like to keep in some form, as it gives us a very high level view of the project consider system. Thus the solution to this problem will be to do multiple clusterings with different filters and therefore also different sets of attributes. This will enable us to cluster attributes at similar levels of abstraction, while retaining the ability to consider all such clusterings when generating models.

With these new clusterings, some interesting results are revealed

there is an actual difference between `general` and `no_language` clusters, talk about it -> compare images of general cluster. Two options now lay before us, the first would be to stubbornly stick to the 80% limit and try to create more and more specific filters, the other one would be to try to explore lower limits Upon thinking about it further, the 80% may not actually make much sense, as it intentionally hides elements, which would have been otherwise shown

our goal here is to divide files into clusters in which these files would present a simmilar functionality

writing ideas -> show diversity chart for no languages and general clusters, talk about how you found big list of html files and a big list of many java files with relatively low diversity

As it turns out, quite a few attributes besides detectors can be converted to genomes. Let us for example take the structure of an xml file see issue 227

Genomes version one -> failed -> we wanted to turn files into genomes, cluster them and extract diagrams based on most common attributes in each cluster -> this failed as no good filters are available (bad detectors clutter clustering process) and we are mixing detectors in different abstraction contexts Genomes version two -> good filters, many clusterings

The solution to this problem, is to introduce the concept of detector context. This will present us with a way of grouping together detectors which operate in a similar context. If we were to for example take the build context

Secondly, what might also be of some interest is the phylogenetic tree, which can be seen in figure 3.4 TODO

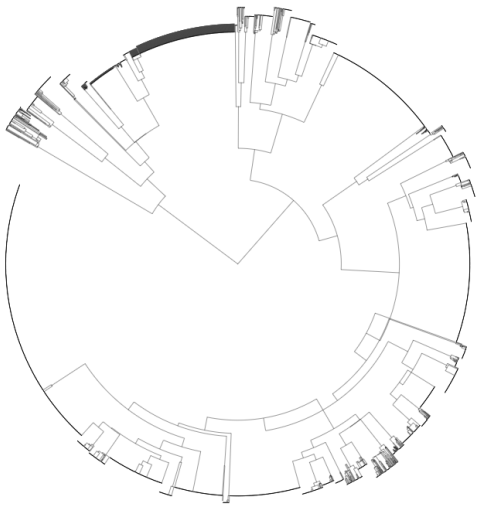


Figure 3.4: Phylogenetic tree generated out of our genomes

3.6.3 Conclusions

3.7 Abstraction Context

3.7.1 Idea

3.7.2 Implementation

3.7.3 Conclusions

Chapter 4

Results

4.1 Summary

Conclusion

Bibliography

[1] Anquetil, N. et al. Modular Moose: A New Generation of Software Reverse Engineering Platform. *Ben Sassi, S., Ducasse, S., Mili, H. (eds) Reuse in Emerging Software Engineering Practices. ICSR 2020. Lecture Notes in Computer Science()*, 12541, 2020. Springer, Cham. `https://doi.org/10.1007/978-3-030-64694-3_8`.

[2] Felix Bachmann, Len Bass, Charles Buhman, Santiago Comella-Dorda, Fred Long, John Robert, Robert Seacord, and Kurt Wallnau. Volume ii: Technical concepts of component-based software engineering. Technical report, Technical Report CMU/SEI-2000-TR-008, Carnegie Mellon Software Engineering . . . , 2000.

[3] Felix Bachmann, Len Bass, Paul Clements, David Garlan, James Ivers, Reed Little, Robert Nord, and Judith Stafford. *Documenting Software Architecture: Documenting Interfaces.* 08 2002.

[4] Carliss Y. Baldwin and Kim B. Clark. *Design Rules: The Power of Modularity Volume 1.* MIT Press, Cambridge, MA, USA, 1999.

[5] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice.* Addison-Wesley Professional, 3rd edition, 2012.

[6] Martin Beck, Jonas Trümper, and Jürgen Döllner. A visual analysis and design tool for planning software reengineerings. In *2011 6th International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT)*, pages 1–8. IEEE, 2011.

[7] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture - Volume 1: A System of Patterns.* Wiley Publishing, 1996.

[8] Choongki Cho, Ki-Seong Lee, Minsoo Lee, and Chan-Gun Lee. Software architecture module-view recovery using cluster ensembles. *IEEE Access*, 7:72872–72884, 2019.

[9] Remco C de Boer, Patricia Lago, Alexandru Telea, and Hans van Vliet. Ontology-driven visualization of architectural design decisions. In *2009 Joint Working IEEE/IFIP Conference on Software Architecture & European Conference on Software Architecture*, pages 51–60. IEEE, 2009.

[10] Stephan Diehl. *Software visualization: visualizing the structure, behaviour, and evolution of software*. Springer Science & Business Media, 2007.

[11] Daniel Dvorak. Challenging encapsulation in the design of high-risk control systems. In *Proceedings of the 17th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'02)*, pages 87–99, 2002.

[12] SD Eppinger. *Design Structure Matrix Methods and Applications*. MIT Press, 2012.

[13] Joshua Garcia, Igor Ivkovic, and Nenad Medvidovic. A comparative analysis of software architecture recovery techniques. *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 486–496, 2013.

[14] T.O. Group. *ArchiMate® 3.2 Specification*. The Open Group Series. Van Haren Publishing, 2023.

[15] R.S. Hall, K. Pauls, S. Mcculloch, and D. Savage. *OSGI IN ACTION, CREATING MODULAR APPLICATIONS IN JAVA*. Wiley India Pvt. Limited, 2011.

[16] Anton Jansen and Jan Bosch. Software architecture as a set of architectural design decisions. In *5th Working IEEE/IFIP Conference on Software Architecture (WICSA'05)*, pages 109–120. IEEE, 2005.

[17] Musengamana Jean de Dieu, Peng Liang, Mojtaba Shahin, Chen Yang, and Zengyang Li. Mining architectural information: A systematic mapping study. *Empirical Software Engineering*, 03 2024.

[18] Jung Soo Kim and David Garlan. *Analyzing architectural styles*, volume 83. 2010. SPLC 2008.

[19] Frederick Kuhl, Richard Weatherly, and Judith Dahmann. *Creating computer simulation systems: an introduction to the high level architecture*. Prentice Hall PTR, 1999.

[20] Marc Lankhorst et al. *Enterprise architecture at work*, volume 352. Springer, 2009.

[21] Manny M Lehman. Laws of software evolution revisited. In *European workshop on software process technology*, pages 108–124. Springer, 1996.

[22] Spiros Mancoridis, Brian Mitchell, Yih-Farn Chen, and Emden Gansner. Bunch: A clustering tool for the recovery and maintenance of software system structures. *Conference on Software Maintenance*, pages 50 – 59, 02 1999.

[23] O. Nierstrasz and S. Demeyer. *Object-oriented reengineering patterns*. 2004.

[24] OMG OMG. Unified modeling language (uml) specification version 2.5.1, 2017.

[25] Thomas Panas, Thomas Epperly, Daniel Quinlan, Andreas Saebjornsen, and Richard Vuduc. Communicating software architecture using a unified single-view visualization. In *12th IEEE International Conference on Engineering Complex Computer Systems (ICECCS 2007)*, pages 217–228. IEEE, 2007.

[26] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, December 1972.

[27] James Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified Modeling Language Reference Manual, The (2nd Edition)*. Pearson Higher Education, 2004.

[28] Mojtaba Shahin, Peng Liang, and Muhammad Ali Babar. A systematic review of software architecture visualization techniques. *Journal of Systems and Software*, 94:161–185, 2014.

[29] Mary Shaw and David Garlan. *Software architecture: perspectives on an emerging discipline*. Prentice-Hall, Inc., 1996.

[30] Fabian Sievers and Desmond G Higgins. *Clustal Omega, accurate alignment of very large numbers of sequences*. Springer, 2014.

[31] Ian Sommerville. *Software engineering. 10th*, volume 10. Addison-Wesley, 2015.

[32] Richard N Taylor, Nenad Medvidovic, and Eric M Dashofy. *Software architecture: foundations, theory, and practice*. Wiley Publishing, 2009.

[33] V. Tzerpos and R.C. Holt. ACDC: an algorithm for comprehension-driven clustering. *Proceedings Seventh Working Conference on Reverse Engineering*, pages 258–267, 2000.

[34] Vassilios Tzerpos. *Comprehension-driven software clustering*. PhD thesis, 2001. AAINQ63614.

[35] Andrzej Zalewski, Szymon Kijas, and Dorota Sokołowska. Capturing architecture evolution with maps of architectural decisions 2.0. In *Software Architecture: 5th European Conference, ECSA 2011, Essen, Germany, September 13-16, 2011. Proceedings 5*, pages 83–96. Springer, 2011.

[36] Yiran Zhang, Zhengzi Xu, Chengwei Liu, Hongxu Chen, Jianwen Sun, Dong Qiu, and Yang Liu. Software architecture recovery with information fusion. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2023, page 1535–1547, New York, NY, USA, 2023. Association for Computing Machinery.