University of Warsaw
Faculty of Mathematics, Informatics and Mechanics

Łukasz Puławski

Methods of detecting spatio-temporal patterns in software development processes

PhD dissertation

Supervisor
prof. dr hab. Dominik Ślęzak

Institute of Informatics
University of Warsaw

May 2022

Author's declaration:
aware of legal responsibility I hereby declare that I have written this dissertation myself and all the contents of the dissertation have been obtained by legal means.

May, 2022	. . . .
date	*Łukasz Puławski*

Supervisor's declaration:
the dissertation is ready to be reviewed

May, 2022	. . . .
date	*prof. dr hab. Dominik Ślęzak*

Polski tytuł rozprawy

Metody wykrywania wzorców strukturalno-czasowych w procesie wytwarzania oprogramowania

Streszczenie

W inżynierii oprogramowania *antywzorce projektowe* to często występujące błędne rozwiązania typowych problemów programistycznych lub architektonicznych, zwykle utożsamiane z pewnymi strukturami w kodzie źródłowym programu. Liczne analizy pokazują wpływ występowania antywzorców na wyższą liczbę błędów i większe koszty utrzymania systemu. Dlatego metody wykrywania i eliminowania antywzorców stanowią istotny kierunek badań.

Większość dotychczasowych badań poświęcona była wyszukiwaniu konkretnych obszarów kodu źródłowego, które same stanowią wystąpienie określonego antywzorca. W niniejszej rozprawie natomiast wprowadzono pojęcie *reguł czasowo-przestrzennych*, które, między innymi, mogą być użyte do wskazania, w których rejonach kodu źródłowego mogą wystąpić antywzorce w przyszłości. Daje to możliwość przeciwdziałania powstawaniu nowych antywzorców, zanim jeszcze zostaną one wprowadzone do kodu źródłowego.

Reguły czasowo-przestrzenne w procesie wytwarzania oprogramowania oparte są o pojęcia: *relacji przestrzennej*, która odpowiada konkretnym zależnościom wyrażonym w jawnych konstrukcjach języka oprogramowania, oraz *relacji czasowej*, związanej z kolejnością wprowadzania przez programistów zmian do kodu źródłowego. W rozprawie przedstawiono efektywną metodę adaptacyjnego wyliczania tych relacji w toku procesu rozwoju oprogramowania.

Przydatność zaproponowanej teorii została empirycznie potwierdzona przez eksperymenty wykonane na repozytoriach kodu siedmiu systemów typu open-source.

Słowa kluczowe: Eksploracja repozytoriów oprogramowania, Wnioskowanie czasowo-przestrzenne, Antywzorce projektowe.

Abstract

In software engineering *design anti-patterns* are commonly used bad solutions for a recurring problem in software design. They are frequently equated with specific structures in the program source code. Ample evidence shows a correlation between existence of anti-patterns and bad properties of the system, such as higher number of defects or higher maintenance and development costs. Therefore, methods of identifying and eliminating design anti-patterns are vital.

A common approach to the problem is to find the exact areas in the source code which constitute specific anti-patterns. In this thesis we introduce the concept of *spatio-temporal rules*, which, among other applications, can be used to predict in which areas of the source code one can expect some anti-patterns to occur in the future. It enables one to prevent introducing new anti-patterns into the source code.

Spatio-temporal rules are based on two concepts: The concept of a *spatial relation*, which corresponds to certain dependencies expressed by specific constructs in the programming language, and the concept of a *temporal relation*, which relates to the order in which changes are implemented in the source code. An efficient adaptive method of computing these relations along with the software development process is presented in the thesis.

The proposed method was experimentally validated with the use of data from version control systems of a few popular open-source projects.

Keywords: Mining software repositories, Spatio-temporal reasoning, Software design anti-patterns.

ACM CCS: Software and its engineering~Design patterns, Information systems~Association rules, Information systems~Data mining, Computing methodologies~Supervised learning, Software and its engineering~Software defect analysis, Computing methodologies~Rule learning.

Acknowledgments

I would like to thank my supervisors prof. Andrzej Skowron and prof. Dominik Ślęzak for inspiration, patience and help in completing this thesis and dr Maciej Bartoszuk for his valuable remarks on this study.
Most of all I would like to thank my wife. Without her generous support this thesis would never be possible.

Table of Contents

Glossary

abstract syntax tree A tree that represents the structure of the code written in a programming language according to its syntax. Each node in the tree represents a formal construct in the source code. 84, 85, 91, 99–101, 106, 114

branch A concept from the source code management system: one of many parallel lines of development of a software system. 21, 181

call graph A type of dependency graph whose edges represent calls between code entities. 30, 31

class A code entity that is contained in the package and contains methods. 8, 9, 22, 23, 28, 29, 94, 99, 101, 106

code churn A temporal measure defined as the sum of added, modified and deleted lines in a given source code entity over a given period. 82

code entity Part of the structure of the source code uniquely identified by its name. 6–9, 11, 22, 23, 85, 86, 94, 107, 109, 113, 137, 170, 180, 182, 235

code smell A structure in the system source code that might potentially have some bad properties similar to a design anti-pattern. 1, 10, 28, 40, 179, 243

commit A transactional operation of applying changes to the source code management system, uniquely identified by its revision. Information about the author, date and the set of source file modifications is assigned to each commit. 10, 20, 21, 33, 59, 60, 78, 89

complex object An object that has a rich, non-trivial structure and is built from other objects (complex or simple). Additionally, it can be described by simple properties. 11, 18, 19, 45, 58, 59, 88, 94

concept A subset of the information system universe. It aims at representing some observable phenomena in the set of objects formally described in the information system. 7, 46

concept drift A phenomenon in mining temporal or sequential data, in which certain statistical properties of an observed process change over time, so that the quality of the prediction model can gradually deteriorate. 66

core The intersection of all reducts – the set of all attributes that cannot be removed from an information system without information loss. 57, 58

data sampling The technique of repeatedly writing to log a state of a complex object – once in a given sampling interval. 10, 58

decision table An information system with one selected decision attribute that, for a given object, determines which decision class it belongs to. Decision classes usually represent concepts. 46, 156, 165, 173, 177, 183

dependency graph A directional, labeled multigraph that represents dependencies between code entities. The labels denote different kinds of dependencies, such as inheritance (dependency between classes) or invocation (methods). 6, 84, 85, 89

descriptive task A type of data mining task in which the algorithm extracts useful knowledge about the nature of analyzed data. The knowledge is implicitly used to build a predictive model for similar or related data sets. 44, 63

design anti-pattern A frequently used, conceptual structure in the software source code that addresses a common design problem in a bad manner. 3, 6, 9, 10, 27, 28, 40, 86, 116, 125, 127, 128, 137, 156, 158, 160, 166–180, 229, 231, 232, 236–243

design pattern A frequently used, conceptual structure in the software source code that properly addresses a common design problem. 10, 26, 86, 116, 180

design rot The phenomenon of gradual degradation of system design. It can be observed in the systems that undergo a long process of software development. 37, 38

event Timestamped information about an atomic change of a structure or properties of a complex object. Events are written to an event log. 17, 33, 44, 58, 59, 80

event logging The technique of writing to log all events that take place during complex object evolution. 34, 58, 59

feature vector A vector with the values of all attributes for a given element from the information system universe. 46

field The lowest-level (along with method) code entity considered in this research. Fields are contained in a class. 9, 23, 94, 101, 105

implicit dependency An actual dependency between code entities which is not explicitly present in the source code. 74, 81

indiscernibility relation An equivalence relation that pairs such elements which cannot be discerned from each other, on the basis of a given set of attributes. 9, 57

information system A commonly used model for representing knowledge about a series of data in machine learning. In this model each object from the universe is described by values of the same attributes. 7–9, 11, 46

Integrated Development Environment A tool used by developers to create the source code of the system under development. Some IDEs can record developer activities and thus provide an additional log of the software development process. 71, 74, 78–81, 167

issue A record stored in the issue tracker. It is characterized by its type (among other attributes), which may be used to distinguish defects/bugs – a sub-category of issues . 20, 22, 165

issue tracker A system used to support the software development process, which stores information about actions taken on the software during its development (addition of new features, defect fixes, etc.) . 8, 10, 20, 22, 34, 70–72, 74, 76, 88, 89, 91, 92, 165, 180, 245

lower bound The lower approximation of a rough set that represents the set of those elements which certainly “belong” to it. 10, 56, 121, 182

main development branch The main branch in the source code management system, which typically is the basis for future development of new features in the software system. 21, 108

main entity Central, most important code entity of an instance of design anti-pattern. 116, 117, 119, 122, 125–128, 131, 134, 135, 159, 180

merge An operation of applying changes done to the source code in one branch to another branch. 21, 108

method The lowest-level (along with field) code entity considered in this research. Methods are contained in a class. 8, 22, 23, 29, 94, 101, 105

ontology Formal naming and definition of the types, properties, and interrelationships of the entities that exist in a particular domain of discourse. 88

overfitting A situation when a predictive model is too specific so that it has good quality measures only on the training set. 9, 53, 54

package A code entity that groups classes. 22, 29, 94, 136

predictive task A type of data mining task in which the algorithm produces a predictive model for identical or related data which it was trained on . 44, 63

pruning The process of simplification of the classifier - one of the possible ways of coping with the problem of overfitting. 54, 166

reduct The minimal subset of the set of attributes of an information system (A) that induces the same indiscernibility relation as A. It represents the minimal set of attributes that express complete knowledge about objects in the information system. 7, 57

refactoring A process of improving the quality of software structure without changing its external behavior. 36, 37, 77, 79, 80, 82, 83

release The act of submitting a ready-to-use version of software for broader use by target users. Usually preceded by a longer process of development, testing and stabilization. 33, 74, 78, 81

revision A unique identifier of a commit. 6, 10, 20, 21, 33, 34, 168, 180, 181

rough sets A formalism that allows one to represent vague concepts by providing their upper and lower approximations in the form of the upper bound and the lower bound. 54, 56, 57, 63, 183

rough software pattern The concept of a static software pattern expressed in a vague manner by its lower and upper bound. 15, 38, 183

SCM The Source Code Management is usually the central system which stores the current version of the software source code and all modifications done to it in the past. 10, 22, 34, 59, 70, 72, 74, 76, 80, 81, 88, 92, 97, 108, 115, 126, 165, 174, 180, 245

SCM hook A small script run whenever someone attempts to submit a commit to the SCM. Among other things, it allows one to enforce proper SCM and issue tracker synchronization by rejecting commits without a reference to a valid issue in the issue tracker. 72

snapshot The state of a complex object recorded at a certain point in time. Used to store temporal data in data sampling. 58

software evolution The process of change in the structure of the software system that takes place during its development. Formalized as a sequence of software snapshots. 18, 34, 177, 180–182

software pattern A fragment of software structure that can be e.g. a design pattern, a design anti-pattern or a code smell. 33, 34, 153, 156

software snapshot The state of the software source code at a given revision, formalized as a multigraph. 10, 110, 128, 171, 177, 180, 181

source code decay Gradual degradation of the quality of the system source code. It can be observed in the systems that undergo a long process of software development. 38

source code metric A real-valued function defined on a subset of the source code entities, which is used to assess their quality. 23, 82, 83, 86, 128

source file An elementary unit of the source code seen as a text - a single file may contain definitions of one or more code entities. 19, 94, 98, 107, 116

spatio-temporal pattern An observable phenomenon in the course of evolution of a complex object, where spatial patterns appear, disappear or change. 1, 4, 11–13, 15, 19, 49, 92, 93, 103, 152, 169

spatio-temporal relation A relation between two structures that change over time, such that we can define their spatial relation and temporal relation between their occurrences. 3, 4, 34, 40, 80, 143–146, 152, 153, 155–158, 160–163, 167, 168, 170, 175, 177–179, 181, 183

spatio-temporal rule A formal rule that describes spatio-temporal patterns. 15, 40, 41, 152, 154–156, 159, 165–168, 171–174, 177, 178, 183, 236, 241

technical debt A situation when software system architecture is not maintained, so that it gradually deteriorates during the implementation of new features, producing greater maintenance and development costs. Also interpreted as the cost of restoring the quality of software architecture. 38

temporal pattern An observable phenomenon in the course of evolution of a complex object – particularly in the evolution of a software system. 13, 18, 33

time series A series of numeric values, usually built in the process of repetitive storage of a measured quantity during the observation of a process. 44, 60–62, 65

universe A set of elements in the information system. 46, 47

upper bound The upper approximation of a rough set that represents the set of those elements which may "belong" to it. 10, 56, 121, 159, 161, 182

Chapter 1

Introduction

1.1 Spatio-temporal patterns in software development

1.1.1 Patterns in the software development process

Computer systems have not only become a commonplace in virtually all aspects of our lives, but some, very large systems, which have been developed for months or even years, can be considered some of the most complex structures that have so far been created by humans. On a very abstract level, the structure of a software system can be seen as a very large set of substructures, connected by various relations, dependencies and interactions. The network of city streets is a good example of such a structure - we might look at it as a collection of different junctions connected via different types of streets. In such a traffic-flow system we may identify some typical substructures, which consist of various interconnected elements which all play a well-defined role. A roundabout, a cloverleaf interchange junction or a sequence of traffic lights set along the main street are good examples of such substructures. We will call them *spatial patterns* or *static patterns.*

If we add a time dimension to the example above, we can conceive of cars that actually move along the streets. In such a setting we can not only observe some static patterns (e.g. a traffic jam), but also some temporal phenomena that happen over a certain period. A good example is the following: "If cars get stalled in a traffic jam at junction X, then within 5 minutes the average speed at roads S and T decreases by half and within 10 minutes cars at junctions Y and Z get stalled in a traffic jam". We will call such an observable phenomenon a *temporal pattern*.

By analogy, we can identify some spatial and temporal patterns in software. Spatial patterns can be found at any point in time if we view a large structure of a computer system, in particular, its source code. As the system is being developed and its structure is modified so that certain static patterns appear while others disappear, we can observe temporal patterns in this process.

The methods proposed in this thesis focus on the problem of detecting both static patterns in the structure of the software system and the temporal patterns in the course of its development.A spatio-temporal pattern is a term to describe an observable phenomenon that involves changes of static patterns in time.

1.1.2 Spatio-temporal patterns modeling

Research presented in this thesis is mostly based on a strict, mathematical model that describes both static and temporal patterns in the software development process. The strict formalism of the model, however, while appropriate for computer algorithms, is hardly understandable for humans. In such case the potential applicability of proposed methods in real-life situations would be limited. In order to avoid such a drawback, special emphasis is put on the comprehensibility of the model. Generally speaking, the proposed methods for detection of patterns produce extra knowledge on how certain patterns evolve over time during the software development process. The findings are presented in terms of high-level, easy to understand concepts that directly resemble actual phenomena.

Prof. Lotfi Zadeh, when outlining the notion of information granules ([358]), claims that granulation is one of the fundamental concepts in human cognition. One of the elementary observations in this and his later works (e.g. [359]) is the fact that if we want to model a real-world phenomenon that people need to cope with, we must introduce vague, conceptual units of computation and be able to operate on them.

Prof. Andrzej Skowron and dr Piotr Synak have proposed a similar view on the matter, related to a more specific domain of approximate modeling of temporal data ([318]). The proposition yields a high-level general framework for approximate reasoning about spatio-temporal patterns in complex data. Among other observations, we can draw three conclusions about how to properly construct a mining algorithm in such a context. These are: 1) the hierarchical structure of machine learning algorithms, 2) incorporation of some form of expert knowledge to describe complex concepts in the algorithm and 3) the elementary assumption that the model should be based on vague concepts.

The hierarchical structure of machine learning algorithm is derived from the elementary observation that when a human expert tries to identify and describe some patterns in a large complex environment, they do not use elementary level observations, but rather try to express them in terms of a high-level, specialist language.

For example, when we use weather forecast models, we do not derive highest-level concepts, such as “good weather”, or “rainy afternoon”, from the lowest-level measurable facts, such as inter-particle interaction. On the contrary, we describe them by using just lower level concepts such as “overcast”, “humidity” or “wind direction”, and those come from still lower-level concepts such as “air mass movements”, “steam convection” or “heat conduction”. In the end, the lowest-level of this hierarchy contains measurable facts, such as wind speed, humidity or temperature, that can be taken directly from simple sensors. The aforementioned expert knowledge is used to express inter-relations between concepts in subsequent levels of hierarchy and potentially some interactions between concepts from the same level.

The structure of experiments presented in this thesis follows, to some extent, the aforementioned approach. The proposed model, embedded in the domain of software engineering, is based on simple, measurable facts, which are then aggregated to different, more complex concepts on different levels of hierarchy. The vague concepts and the application of expert knowledge is used in different parts of the proposed method.

1.1.3 Software quality

Know-how on how to properly implement software systems is of vital importance and is widely researched, with thousands of scientific and popular publications every year. After decades of experience in the development of large computer systems, a lot has been learned about good and bad practices in this field. Quite surprisingly, it seems that while the latter are still frequently applied to newly developed systems, the former sometimes tend to be forgotten or ignored. Any attempt to better understand why this is the case seems crucial and could allow us to improve the way we work with computer systems nowadays. Research presented in this dissertation can help us better to understand how software is being developed and what the reasons for common problems are. An appropriate model of spatio-temporal patterns allows for predicting where and when bad structures or defects may appear in the source code of a developed software system.

1.2 Main contributions

The problem of detecting static patterns in software structure has been intensely researched in recent years ([102], [175], [302], [321]). A common approach to automation of the detection process is to use a formalism that describes the pattern in a formal language (e.g. [338]) and then expresses the structure of the software system in the same language. Approaches of this kind are discussed in detail in section 5.2.7. This thesis introduces the concept of the *rough software pattern* that copes with the problem of expressing and detecting vague concepts in the software system structure. For more details, please refer to Section 6.3.2.

Analysis of software evolution is another very popular research topic ([295]). The focus is typically on the measurement, visualization and other formal ways of analyzing software evolution, which is in the end used by a human expert in order to identify and describe its temporal patterns (see [116]). The method proposed in this thesis goes one step further: it automates the process of identification and discovery of both spatial and temporal patterns with the use of elementary machine learning algorithms. It is based on the key concept of a *spatio-temporal rule*, and it is specifically fitted to the software development domain, so that it can efficiently produce appropriate models in an adaptive manner when the system is iteratively modified by developers. For details, please refer to Chapter 6.

Defects, such as ill-structured parts of the source code of software systems, tend not to be evenly distributed over all the elements. Some source code elements have a tendency to be more prone to errors. This phenomenon has been widely researched. The proposed methods vary from simple (sometimes even trivial) measurements ([195], [284]), to more sophisticated methods, based on advanced statistical analysis ([233], [71]). A key observation is that in most cases the proposed methods do not give reasons or even indicators of defect proneness, since the general research question posed is "density of what factors is positively correlated with the number of detected defects."

(e.g. [166], [167], [351]). One of the research questions posed in this thesis is formulated differently: “are there any factors that increase the probability of the appearance of defects in the future”. This question is stated more specifically in Chapter 3. Chapter 6 presents a framework that enables us to identify such factors.

1.3 Remainder of this thesis

- Chapter 2 explains concepts in the domain of software engineering and outlines the view on the software development process used in this study.
- Chapter 3 explains the motivation for the research with a special focus on the potential practical applications of the methods described in this thesis in real-world software development tools. It also outlines research goals and hypotheses of the present thesis.
- Chapter 4 introduces and - if necessary - formalizes fundamental concepts and notions that are necessary to understand this study.
- Chapter 5 discusses existing literature and methods in similar research fields. It focuses on both the broader context of mining spatio-temporal data in general and the specific context of mining data from the software development process.
- Chapter 6 introduces the pattern mining framework in detail. This chapter describes the main contribution of this thesis and explains how it was validated experimentally on data gathered from real-world software projects.
- Chapter 7 summarizes the research done for this thesis, draws conclusions and discusses possible future research.
- Appendix A is a report from the execution of all the experiments referred to in this thesis. It contains information about data, the method and the result of each experiment. Specific experiments described in this Appendix are individually referred to from other chapters.
- Appendix B gives instructions on how the experimental results described in this thesis can be reproduced.

Chapter 2

Software development basics

This chapter describes the conceptual domain of the study, by attempting to characterize

- What the domain of the present thesis is: the evolution of complex structures in time.
- How software development process is organized, how data about its execution is stored in various systems. What events can be recorded during execution of the process.
- What the patterns are in general (both static and temporal) and how they can be represented and mined from data.
- What kind of patterns (static, temporal) can be present in the software development process.

Formal definitions of these concepts are given in Chapter 6. A reader familiar with the topic of software engineering and mining software repositories may skip directly to chapter resume in Section 2.5.

2.1 Evolution of complex structures

Before we look into the particular domain of the software development process, we need a more general view on the specific kind of data used and the research methods.

A *complex object*[1], as opposed to a *simple object* is something that has an elaborate, non-trivial structure and is built from other objects (complex or simple). To give an example, we could say that a car is a complex object, which consists of many other objects, bound together in such a way that they form a vehicle. The engine is a specific part of the car, but it is itself again a complex object, whereas a single screw – in most cases – is a simple object. Each object (complex or simple) has its own characteristics that can usually be expressed in terms of numeric, textual or symbolic attributes. In the above example the car is of a certain color, the engine has a certain displacement and the screw has its diameter. What is especially important in the case of a complex object, is the fact that the structure of how its parts are related to each other can also be described by certain attributes. The way in which the engine is mounted in a car (longitudinal or transverse engine) is neither an attribute of the car body nor of the engine, but rather it is an attribute of their relation.

A rich/elaborate structure, such as a complex object, can change over time. We will call such a process the *evolution* of the object. The life of any living organism, e.g. the life of a human being, is a good example of the evolution of its body. The human body is a complex object that is built of organs, which in turn are built of tissue, etc. In a lifetime, the body changes in many different ways, including growth, aging, interaction between different kinds of tissue or organs. We can observe different temporal phenomena or processes in the evolution of the body, such as sleep (short-term) or growth (long-term). We will call them *temporal patterns*.

Observable effects of the patterns can be measured thanks to attributes. During the evolution of a complex object, its constituents may change, appear or vanish. Since objects, their structure and their relations, can be described in terms of their attributes, we may perceive the evolution of a complex object as a sequence of values of the attributes. For example, if we take the weight and height as attributes of the body, the pattern of growth will be reflected in their continuous increase.

Software constitutes a perfect example of a complex object, and its development can be treated as its evolution. We will therefore call this process *software evolution*. Moreover, a fragment of a software, which is modified

[1]Please note that in this context we are not referring to the concept of the *object* in the object-oriented programming, but to the general concept of the arbitrary object with well-defined inner structure.

over time may also be considered a complex object with its own evolution. Consequently, the problem of finding spatio-temporal patterns in the software development process is a particular case of finding temporal patterns in the evolution of a complex object.

A formalized model which defines the structure of software and its parts that are being developed is given in Chapter 6:

Section 6.1.1 introduces the concept of *software snapshot*, which embodies the structure of the software source code as a formal complex object.

Section 6.3.1 defines the concept of *pattern instance*, which can be viewed as a complex object corresponding to the fragment of the software.

The evolution of such structures is further discussed in Section 6.5.

2.2 Software development process

Software development is a long-lasting process, which involves many programmers, working together on the development of a computer program. One can consider this process from a variety of perspectives, including, multi-agent, social or interaction-centric models.

Firstly, it can be seen as a *multi-agent system*, where the developers are autonomous *actors*, working in a common *environment* and *interacting* collaboratively according to a certain *plan*, in order to achieve their common *goal*. Since the agents, who take part in the process, are real human beings, each with a different background, culture or character, the process is also often investigated as a *social phenomenon*, with special focus on the inter-human *interactions*, where psychological aspects count as much as expert knowledge in the field of software engineering. The *functional* view on the process is more abstract and high-level. In this perspective, one can see a system as a black box that provides certain functionalities. During the development the set of functionalities is modified and we only focus on these changes. Last but not least, one can see the software development process only as data (i.e. the source code, documentation, etc.) that is created and persists during its execution, with no reference to the way in which this data was created or what the goal of creating it was. We will call such a view a *technical* perspective.

The present thesis takes into account only the technical view. To better understand the difference, we may take a look at a simple example: When two developers change a source file, one shortly after the other, assuming the technical perspective, we are only interested in the modifications of the file, with special focus on how the structure of the developed program was changed. We are not interested in how the two people interacted with each other before, whether they exchanged some communication, what their intention was, etc. The technical view on the software development process is explained in more detail in the sections that follow.

2.2.1 Issue tracker (IT)

The *issue tracker* (*IT* for short) is a system which is used to support the software development process. It stores information about actions taken on the software during its lifetime, as seen from the *functional* point of view. A single action is called an *issue*. The issue has its own specific *type* and is supposed to contain all the relevant information about what needs to be implemented in the system in order to consider the issue complete.

Types of issues used in different systems vary, but there is always one special kind, called the *bug*, which describes a known defect in the system. An issue of such type can be considered finished (or closed) when the defect is fixed.

The issues in IT have their life-cycles, which consist of several steps, such as creation, assignment to the person in charge of it, resolution of the problem and, eventually, closure of the issue. All such actions are recorded by IT. Therefore, since all tasks must be recorded in detail by IT, one can treat it as a log of the history of system development.

2.2.2 Source Code Management (SCM) system

Source Code Management System or *Source Control System* (*SCM* for short) is usually a central system which stores the current version of the software source code and all modifications done in the past. It allows the developers to apply their changes to a common source code base in a transactional manner. Such modifications are called *commits*. A commit has a unique identifier, called *revision*, which might be just a subsequent number in a sequence. Moreover, it contains the information about timestamp, the information about the author (called *committer*), a short textual message entered by him/her and a list of modifications applied to many files managed by the SCM. Technically, the commit also contains information about the commit(s) it directly follows, so it is possible to arrange the commits into partial order. Therefore, the log of SCM can be seen as a history of the software development process, as seen from the technical perspective mentioned in the preceding section.

2.2.3 Branches and merges

Most real-world software systems are developed simultaneously in many parallel versions, called *branches*. Typically, there is one main branch (we call it the *main development branch*), which contains the most recent development version of the system. Such a version incorporates important changes that are implemented and new features which are added, therefore, the system build from it may become unstable. Stable versions are usually located in dedicated branches and only small improvements and bug fixes are applied there. Still, on some occasions changes done in one branch need to be applied also to another branch or branches. Such an operation is called a *merge*.

In this research, if not stated otherwise, we assume time to be linear and, consequently, only analyze software development process in one branch, technically the main development branch. We also assume that commit done to the main development branch cannot be later deleted or modified in any way (e.g. by squashing commits – see [61]). Please note that, since every branch appears initially as a copy of another branch, the period of analysis in linear time can be longer than the life of a branch and can actually span over more branches. Technically, it means that the analyzed fragment of the software development process can be uniquely identified by a linearly-ordered set of revisions. This concept is depicted and explained in figures 2.1.

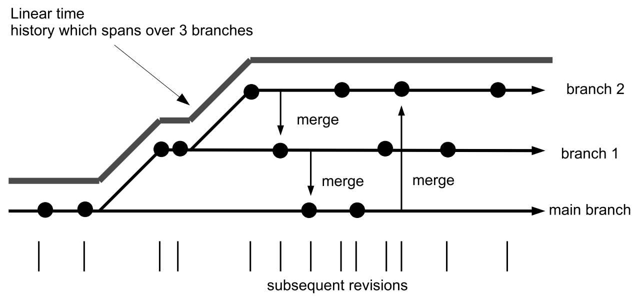

Figure 2.1: The concept of branches and merges in the source control management system. Commits are denoted by black spots and subsequent revisions by vertical lines at the bottom.

2.2.4 Synchronization between SCM and IT

In Section 2.2.2 we concluded that logs of the SCM can be seen as a technical track of the process of software development. Similarly, as mentioned in section 2.2.1, the issue tracker logs can be seen as the track of the same process, but from a functional point of view. The question arises how to synchronize these two views. More precisely: given that issue tracker contains a collection of functional tasks done in the system, and SCM contains a collection of changes in its source code, can we tell what changes in the source code were necessary to complete a particular issue tracker task, or what issue tracker task was the specific motivation for a particular change recorded by SCM? There are many approaches that can be taken. Some of them are roughly described in Section 5.2.1. It is common practice to put the unique identifier of a related task from the issue tracker to the commit message. By doing so, the developers bind all changes in SCM to issues recorded in the issue tracker and consequently provide a virtual synchronization of two different views of the software development process. For the experiments described in the thesis that are based on both sources of process log, it is assumed that the aforementioned practice is always applied and therefore synchronization between issue tracker and SCM is taken for granted. However, most methods and experiments are based solely on the logs from SCM. In such case the requirement of having the synchronization in place can be omitted.

2.2.5 Software structure

So far we have assumed that a version of software is a collection of source code files. Obviously, the contents of the files have certain semantics that represent the formal structure of the program in a particular programming language. On a general level, we assume that the structure of the software is a graph, where nodes represent implemented logical elements of the source code and edges represent different dependencies between them. The elements will be called *code entities*. Examples of code entities are *packages*, *classes*, *methods* or *procedures* and dependencies can be exemplified by *calling*, *using*, *including* or *extending*.

These graphs are further discussed in Sections 2.2.6 and 2.2.11.

2.2.6 Object-oriented programming

Object-oriented programming is a paradigm in which the structure of the source code consists of *objects*, which are instances of *classes*. Each object stores its own data in *fields* and provides some functions via its *methods*. Both fields and methods are defined in the class, so if two object are instances of the same class, then they have identical sets of fields and methods. Each class can *inherit* from another class (or classes)[2]. In such case the inheriting class derives fields and methods from the class it inherits from. The relation of inheritance cannot contain cycles. This yields a hierarchical structure of inheritance between multiple classes.

The objects interact with each another by *calling* their methods. If one method calls another we can consider that they are connected by some kind of edge. Therefore, the core structure of the software system implemented in an object-oriented approach can be seen as a multigraph where nodes represent objects and edges represent their interaction. This concept is further extended in Section 2.2.11.

2.2.7 Software source code metrics

Each code entity is defined only by its own source code, which formally is a string[3] from a formal language corresponding to the programming language.

Source code metrics (see [206]) are formally real-valued functions defined on the set of source codes that correspond to a certain subset of code entities. Sometimes, for the sake of clarity, we can treat them as the functions defined directly on the subset of code entities. Therefore, if c denotes an entity, $Source(c)$ - its source code and M - some metric, the value of metric M for code entity c is formally $M(Source(c))$, however, the $Source$ function is sometimes omitted and $M(c)$ notation is used instead. In the present thesis we only consider metrics defined for files, classes and methods. The purpose of using metrics is to quantitatively measure certain quality properties of the code entities. In most cases, the metrics express complexity of the measured entity. If for a given entity the value of the metric is very low, then the source code of the entity can be considered trivial. If the value is very high, then the

[2]In relation of inheritance, we will call the inheriting class a *subclass* and the class it inherits from a *superclass*.

[3]Some entities such as packages do not have a source code. For formal model please refer to Section 6.1.

source code of the entity can be considered too complicated. More generally, we might say that each metric has a set of *proper values* and *improper values*. Then we might say that if for a given code entity c, the value of a metric M is in the proper values set, then c is well implemented with respect to metric M. On the other hand, if $M(c)$ is in the set of improper values, we say that c is poorly implemented with respect to M.

Typically, when the metric represents a complexity of the measured entity, the proper and improper values sets can just be considered separate intervals of $\Re$. In order to simplify the matter even further, we can give a real value Thr_M, which is a threshold that partitions $\Re$ into two such intervals. In such case we will say that c is well implemented with respect to M iff $M(c)$ is below Thr_M and c is poorly implemented iff $M(c)$ greater than Thr_M.

In both cases, such a strict division into good and poor implementation can be controversial, especially, when we use a single metric, which is not a very sophisticated tool. Therefore, in the case with Thr_M, it is better to assume that entities whose value of metric M is close to Thr_M, cannot be reliably assessed as poorly- or well-implemented. Moreover, the specific threshold may vary slightly depending on the characteristic of the analyzed system (see [352]). Therefore statement on the quality of implementation is sound only when the value is significantly below or significantly above the threshold. This phenomenon is discussed in detail and formally defined in section 6.3.2. Here we only focus on one example:

Suppose that M is a simple metric which denotes the number of lines of code in a given source file. If, based on expert knowledge, we set a crisp threshold for this metric at 500 lines, then a file with 499 lines is considered well-designed and a file with just 2 lines more is considered to be too large. Such an approach is obviously too strict. When an experts provide such a threshold, they rather think that a source file which has significantly less than 500 lines is of appropriate size, the file with significantly more than 500 lines is overcomplicated, but we cannot judge on the complexity of such file if its size is close to 500 lines.

In the following paragraph you will find some examples of popular software metrics with a short explanation and a rationale for using it. These particular metrics are used in the formal model described in Chapter 6.

Fan-out is a metric applicable to any code entity. It measures the number of other entities that the measured entity depends on. When a measured code entity depends on many other entities, it is likely to be affected by any change done in them. Therefore in a well-designed software system, the value of this metric should be low.

NPath complexity is a metric applicable to any entity that contains a runnable block of code. It measures the number of all theoretically possible acyclic paths along which the execution of the block can go, taking into consideration all branching instructions in the block. The value of this metric increases with the complexity of the measured block. When it is too high, the block is overcomplicated and hard to understand or maintain.

Cyclomatic complexity is a much simpler approximation of the NPath complexity. Instead of measuring all theoretical possible paths, it simply measures the number of branching instructions of the measured block.

NCSS stands for Non Commenting Source code Statements and it roughly represents the number of actual source code lines in a measured code entity. This metric is not sensitive to different formatting or comments in the source files - the lines of code are always counted in a normalized way. The metric is also called **LOC** - Lines of Code. It is applicable to all code entities.

Object-oriented software metrics

Since object oriented programming is a popular and widely-used paradigm, suites of especially fitted software metrics have been created. This section contains examples of popular object-oriented metrics briefly characterized.

Data abstraction is a metric applicable to class and method. It measures the number of instantiations of other classes in the measured code entity. When the number of such instantiations is high, then the data structure used in the measured fragment of the source code is complex, which makes it hard to understand and maintain.

Depth of inheritance tree is a metric defined for a class. It measures the number of ancestors of a given class in the class inheritance hierarchy. When such a hierarchy is too high, then it is hard to understand the responsibility of a class, since its parts are spread over many ancestors.

Number of fields is a metric defined for a class. It measures the number of fields the class possesses. When a class has too many fields, it operates on data structure that is too large, is harder to understand or maintain and may break the single-responsibility principle.

Number of methods metric is similar to the number of fields. However, it measures the number of methods instead of the number of fields. The rationale is also similar, but instead of measuring how much data a class possesses, it measures how many functions it provides.

Computing of metrics

When we look at the structure of a source code as a collection of interdependent and collaborating source code entities, we may divide software metrics into two categories:

- *entity-centric*, which concern only a single entity and can be evaluated on it without knowing the context of other entities that are somehow related to it.
- *relation-centric*, which concern relation or collaboration between different entities. Here we have to analyze the structure of dependencies around the measured entity in order to evaluate the value of the metric.

There is a practical difference between these two types of metrics: the former is not related to the graph of inter-entity dependencies and therefore it can be computed solely on the basis of the source code of this entity without referring to other entities. This difference is further explored in section 6.1.1, where the formal model is defined in detail. This also has implications on the complexity of the adaptive algorithms described in Section 6.3.5.

2.2.8 Design pattern

Design pattern is a generic and reusable solution for typical problems in software design. Design patterns are not strict formalisms or ready-to-use parts of a system source code. They are rather conceptual ideas and guidelines on how to cope with a commonly occurring problem. In the context of object oriented design, the pattern usually describes a set of collaborating objects, with their roles, relationships and dependencies.

Generally speaking, the design patterns can be categorized into three types ([125]):

- Creational design patterns - used in the aspect of creation and initialization of objects during software run-time.
- Structural design patterns - used in the aspect of a static relation between collaborating objects.
- Behavioral design patterns - used in the aspect of communication between entities during program execution.

A few examples of design patterns with short intuition and rationale for using them are described in the following paragraphs.

Factory method is a creational pattern that allows one to create objects without knowing their concrete class. The factory pattern encapsulates the process of object creation, since it may potentially be complex and depend on context, system configuration and other factors. Moreover, this pattern reduces coupling between the creator and the created objects ([125]).

Adapter is a structural design pattern which enables one to connect two incompatible elements of software, by introducing an intermediary element that can communicate with both of them and translate messages from the sender to a form comprehensible for the receiver ([125]).

Observer is a behavioral pattern that allows one to broadcast information about changes of the state of an object to other objects (observers), without introducing a hard, static dependency between them ([125]).

2.2.9 Design anti-pattern

Design anti-pattern can be seen as a dual concept to the design pattern. By definition it is a commonly used, bad solution for a recurring problem in software design. Software developers, when faced with a common design problem, tend to reinvent some solutions which are well-known for their bad properties and widely discussed in available literature. This phenomenon definitely has many sophisticated reasons, which are discussed in a variety of scientific and popular publications ([64]). While the exact causes are beyond the scope of the present thesis, we attempt to find good indicators for the appearance of some anti-patterns. More details are given in Chapter 3.

The following paragraphs contain a few examples of design anti-patterns with a short intuition and rationale for using them.

God superclass also known as **Base bean** is an anti-pattern in object-oriented design, in which the base class is a collection of numerous utility methods, used by its subclasses. Such a design breaks some fundamental concepts of object-oriented programming: The relation between subclass and superclass does not resemble the actual domain model, the superclass most probably has many responsibilities and it usually does not store any state useful for subclasses ([197]).

Brain class is an anti-patten in object-oriented design, in which a class exposes much complex functionality and does not delegate its parts to other classes. Conceptually, such a class is a central, over-complicated processing unit of broad functionality of the system. Such sophisticated code entities tend not to be extendable and are hard to understand ([259]).

Circular dependency is a situation in which two or more unrelated code entities are mutually dependent on each other. Such a tight coupling is an indicator that a single entity cannot be (re-)used separately, and any change in either of them can cause a *domino effect* ([365]).

2.2.10 Code smells

Code smell is a property or a structure in a system source code that might potentially have some bad characteristic just like a design anti-pattern ([226]). In some sense code smells might be considered "light" anti-patterns or early stages of development of an anti-pattern. The latter interpretation is especially interesting when you consider temporal evolution patterns of code structure. In some studies code-smells and anti-patterns are treated uniformly (see [180]).

The following paragraphs contain a few examples of code smells with a short intuition and rationale for using them.

Refused bequest is a situation in which inherited classes override some methods, without keeping their upward compatibility ([210]). In such a case the Liskov substitution principle is violated ([213]).

Large method/class simply denotes a method (respectively: class) that is very large and therefore hard to understand and maintain.

Method with too many parameters is a method that requires a large (too large) number of input parameters. Such a definition of method indicates that it either encapsulates very complex functionality or it does not use all inputs provided in the parameters.

2.2.11 Graphs in software modeling

In the preceding sections we have mentioned that graphs can be used to model certain structures in the software source code. We will now focus on graphs that are frequently used (in particular in this thesis) for modeling software structure, behavior or semantics.

Dependency graph

Please recall that the logical structure of the source code is built from elements called code entities. In object-oriented software, the code entity might be a package, a module, a class/object or a method. Clearly, such entities are related to each other and there are many types of such relations.

Firstly, there is the relation of *containment*. It models the fact that one entity is contained in another, e.g. classes are contained in packages and methods are contained in classes. Please note that such a relation yields a hierarchical structure of containers and elements. In our example packages are the first level of hierarchy, classes - the second and methods - the third.

The second type of relation is a general *dependency*:

Definition 1. *We will say that entity e_1* depends *on e_2, iff e_2 is necessary for e_1 to compile and work in a running program.*

In other words, if e_2 disappears, then e_1 neither can be compiled nor work properly during program execution. For example, if source code of class c_1, declares that it requires class c_2 to work, but actually, c_2 is never used by c_1

during program run time, then c_1 is still dependent on c_2. Examples of such dependencies are given below:

- If the source code of one class contains a reference to another class, then these two classes are dependent by the *reference* relation.
- If one method calls another method, then they are dependent by the *calling* relation.
- If one class extends another class, then they are dependent by the *inheritance* relation.

The above examples show that there are different types of dependencies between code entities. In order to model such a structure properly, we need to use a multigraph - a graph with possibly multiple labeled edges between two nodes. It can also be considered a *property graph* (see [34]). For the sake of simplicity, we will use simple graph terms in the following section, while keeping in mind that two nodes can be connected by more than one edge. In the following paragraphs you will find definitions of different dependency graphs used in this research.

Definition 2. *For a given version of the software source code, the* dependency graph *is an edge-labeled multigraph* (V, E)*, such that* V *is a set of all code entities in this version of software and* $(e_1, e_2, t) \in E$ *iff* $\{e_1, e_2\} \subseteq V$ *and* e_1 *depends on* e_2 *and* t *is the type of this dependency.*

The dependency graph, defined above, is the most general model of dependencies between code entities. The following sections contain definitions of more specialized dependency graphs.

Call graph

Definition 3. *For a given version of software source code, the* call graph *is a graph* (V, E)*, such that* V *is a set of all methods in this version of software and for any two methods* $e_1, e_2 \in V$ $(e_1, e_2) \in E$ *iff* e_1 *calls* e_2*.*

The above definition refers to a graph that connects methods by a relation of being called. This primitive construction directly follows the structure of the source code: the call from one method to the other, is directly placed in the source code of the calling method. Please recall from section 2.2.11 that code entities can be arranged in a hierarchical structure by the relation of containment. This allows us to define a call graph on any level higher than the level of methods. For example, we can say that class c_1 *calls* class c_2 iff any method contained in c_1 calls a method contained in c_2. This yields the definition of generalized call graph:

Definition 4 (generalized call graph)**.** *Let $G = (V, E)$ be a call graph, let relation C be defined as follows: for any $e_1, e_2 \in V$, $e_1 C e_2$ iff e_2 is contained in e_1 or $e_1 = e_2$. Let C^* be a transitive closure of C.*
The generalized call graph*, derived from (V, E) is a graph (V', E'), such that $V' = V$, and $(e_1, e_2) \in E'$ iff there is an edge $(e'_1, e'_2) \in E$, such that $e_1 C^* e'_1$ and $e_2 C^* e'_2$.*

Conceptually, the generalized call graph introduces the concept of granular view on the call graph: the elementary facts coming directly from the source code (calls between methods) define the call dependency on a higher level (classes, packages), but also cross levels, e.g. from class to package. Other types of dependencies between code entities can be generalized on the basis of the transitive closure of the containment relation. We will use this method in formal constructs defined in Chapter 6.

Inheritance tree

Definition 5. *Inheritance tree is such a graph (V, E) that V is a set of all classes and for $c_1, c_2 \in V$, $(c_1, c_2) \in E$ iff c_1 inherits directly from c_2.*

Some programming languages allow a class to inherit from more than one superclass or not to inherit from any class. In such case, the above definition yields a directed acyclic graph, rather than a tree. However, in this research, we will focus on programs written in Java, which: 1) enforces that each class inherits (directly or transitively) from java.lang.Object class, and 2) does not allow to inherit from more than one class. In this context, the definition yields a single tree for the complete source code rooted at java.lang.Object.

2.2.12 Spatial relations

Since the structure of the software source code can be represented in graphs, we can consider how some notions of graph theory such as adjacency or paths relate to the original source code. In a given representation, any two subgraphs correspond to specific sub-structures in the source code. Conceptually, we can think that if the subgraphs are distant in the graph (e.g. the length of the shortest path connecting some of their nodes is long or when such a path does not exist), then the corresponding structures in the code are loosely dependent on each other. On the other hand, if the graphs are close (e.g. they have multiple common nodes), then the code structures are likely to be tightly dependent on each other. This idea yields the concept of remoteness and closeness of fragments of the source code, which are formally defined in Sections 6.5.1 - 6.5.3. These two types of relations will be called *spatial relations*.

2.3 Temporal aspects of the software development process

Until now we have focused on a static view on the software system, where no time dimension was considered. This section discusses a temporal aspect of the software development process.

2.3.1 Temporal dimension

There are two views on the time dimension, when analyzing the temporal aspect of software: the development time and the run-time. The former view sees the time during the development of the software system and the latter, during the execution of the program. An example of a temporal pattern observable in the first context is the following:"In 90% cases, when file A is modified by a developer, then file B is modified by the same developer within an hour". When thinking in terms of run-time, an example of a temporal pattern is the following: "When method M_1 from object O_1 is called, then within 3 seconds, method M_2 from object O_2 is called". Analysis of the run-time temporal patterns can be a valuable technique for finding some problems in the source code, e.g. system performance bottlenecks.

In this research we focus on patterns in the software development process rather than software run-time process, therefore when we think about time dimension, it always means the time that passes during the implementation of the system by a group of developers.

We can consider different time intervals for the temporal patterns observed during the software development process. They can be bound to the actual time or to events that happen during the process, such as commits, feature implementations, minor or major system releases. In the first case, we will typically analyze phenomena that happen during a fixed time period (e.g. "What temporal patterns were observed last month/week"). In the second case, we do not limit ourselves to a fixed time, but we define intervals by the number of different events, which are part of the software development process. In this context we can ask the question: "What temporal patterns were observed within the last 100 commits", or "How changes made to the structure of the source code were correlated in 10 subsequent minor version releases." In the present research the temporal dimension is primarily based on a commit, i.e. we consider changes to the software done at each revision.

2.3.2 Temporal software development patterns

In the previous sections we have explained the concept of static patterns in the source code structure, which are, generally speaking, frequently occurring pieces of a larger structure. To some extent, they can be formally expressed in terms of graph theory. We have focused on design patterns, anti-patterns and code smells, as concrete examples of such static patterns. There is one common context for all of them - they do not consider time dimension by any means but they are all part of a single snapshot of the system structure, taken at some point in time.

When considering the time dimension in our analysis of the software development process, we may derive a similar concept to such static patterns. We will call them *temporal patterns*. Conceptually speaking, these are phenomena, observable over a sufficiently long period, that focus on changes done to the software structure during its development process. "Every class that is an instance of god class has 75% chances to become a brain class after being modified 100 times", is a good example of a temporal pattern in the software development process.

Lifespan of a pattern Once a software pattern is introduced to the source code of a system at a certain commit, it will probably still be present for some time afterward. When the source code is modified with a commit at a given revision and we can still observe the pattern, we can say that the pattern was present at this revision. The set of all revisions in which the specific software pattern is present in the source code will be called its *lifespan*. A lifespan can be partitioned into maximum intervals of consecutive revisions. Such intervals will be called *occurrences*. Two different occurrences of different software patterns might relate to each other and this is a purely temporal relation. For example, if a given instance of a brain class is present in the first and the second revision only, and then another god class is observed in the fifth and the sixth revision only, we can clearly say that the only occurrence of the former happened *before* the only occurrence of the latter. This concept, together with the above-defined *spatial relation* is used to define the crucial notion of a *spatio-temporal relation* (see Section 6.5).

2.4 Software evolution

Software evolution is a term used in software engineering to describe the process of system implementation over time. Usually the main focus of research in this area is to understand and sometimes formally describe changes in the structure of software that appear during consecutive releases (see [169]). Still, some researchers focus on modifications done in fixed periods or between subsequent revisions. The last, revision-based approach is also used in this thesis. In this context one of the possible ways of looking at software evolution is to consider it as a sequence of all versions of the system that were subsequently created by the developers. Since the structure of a single version of the software can be represented by a specific multigraph (as described in section 2.2.11), the evolution can be seen as a sequence of such multigraphs. Commits are the only possible points in time in which the aforementioned graphs can change. Therefore, it is natural to assume that the sequence can be indexed by the revisions (i.e. the unique identifiers of commits). This conceptual model is formally described in Section 6.1.2.

2.5 Resume

In this chapter we have introduced the basic concepts related to the domain of this research:

1. We have defined what the software development process is and how SCM and issue tracker are used in it. We have stated that from the perspective of this research, logs from these two systems are the only data used to analyze the process (see Sections 2.2.1-2.2.4).

2. We have explained the concept of software entity and, basing on that, we have defined the notion of software structure, especially the structure of object-oriented software, and have shown how it can be modeled with various multigraphs such as dependency graph, call graph or inheritance tree (see Sections 2.2.5 and 2.2.11).

3. We have introduced the concept of spatial relations between different structures in the software source code and the concept of temporal relation between such structures (see Section 2.2.12).

4. We have stated that we limit our considerations to linear time only, even if we analyze a development process on many branches (see Section 2.2.3).

5. We have explained the notion of software evolution and showed that we will consider it as a sequence of multigraphs indexed by the revisions of commits done at SCM by the developers (see Section 2.2.11).

6. We have introduced the concept of static software design pattern, anti-pattern and software metrics (see Sections 2.2.7 - 2.2.10).

Chapter 3

Motivation, research hypotheses and goals

This chapter discusses practical motivation and research context for this thesis. The motivation is understood as a potential practical application of methods proposed in this thesis to solve real-world problems that arise in software development. The hypotheses and goals are presented as formalized statements of these problems in the context of scientific research.

3.1 Motivation

3.1.1 Discovery of evolution patterns for ill-structured software

Anti-patterns and code smells are considered to be bad structures in software source code. Their properties and the effect on software maintenance have been widely described in literature. Still, they do appear in newly implemented software systems and have a tendency to remain in the code for quite a long time. This phenomenon has been widely discussed in publications which offer a variety of hypothetical explanations for such a state. One of the most frequently mentioned reasons is the fact that anti-patterns are simply difficult to remove (see [365], [165], [122]). Therefore, easier detection of the threat of an anti-pattern and the ability to prevent it would be of great value. One of the methods for improvement of software structure is *refactoring*, described in the next paragraph.

Refactoring is a process of changing the structure of the software source code in such a way that the whole system does not change its behavior (as seen from the outside of the system) but internally the properties of the structure, such as lower complexity, better readability or maintainability, improve. A good example of refactoring is the decomposition of large and complex source code entities into a collaboration of a few smaller and less sophisticated ones.

It is believed that refactoring can be a risky and time-consuming operation, especially when executed late and broadly [64]. Consequently, it seems to be beneficial to detect areas for refactoring as early as possible. This leads us to the first motivation:

(Semi-)automated method for detecting early indicators of bad structures in the source code We might say that refactoring can be described as turning a bad structure of the code into a better one. Usually, developers start such activities late, already when the bad structures are broad, frequent or complex, and thus harder to eradicate. Therefore, a tool that warns about deterioration of the structure of the software early, could help to reduce the risk of building an ill-structured system and reduce the cost of maintenance.

There appears to be an analogy to the mathematical models for weather forecasting: they can be seen as tools that warn in advance about dangerous weather phenomena such as storms: If we know that a storm is expected, we can implement countermeasures to reduce its negative effects before it actually arrives. By analogy, when we think about ill-structured software as a storm, refactoring as a countermeasure, then forecast models are the tools that warn us about storms. We believe that the research presented in this thesis is a good basis to build such a tool, because:

- If we know in advance that the system architecture is drifting into bad direction, we can react and improve it early. This leads to a more efficient process of software quality improvement and a potential countermeasure for the *design rot*.

- We can identify areas for refactoring early. Refactoring started earlier is easier to execute and less time-consuming.

3.1.2 Technical debt - hidden cost of software development

If software architecture is not maintained during the system lifetime, its technical condition tends to deteriorate. This leads to higher maintenance and further development costs. It can be compared to the economical rule that interest for unpaid loans increases. This is why these costs are called *technical debt*.

(Semi-)automated method for helping to reduce the cost of technical debt. Software companies can balance between “getting in” technical debt or “paying it off” in order to reduce development or operational costs in the future. Usually, some predictive analysis is made before taking such strategic decisions. Some methods described in this thesis cope with the problem of detecting temporal patterns related to anti-patterns or code smells and phenomena of design rot and source code decay (see: [108], [165]). We therefore believe that some methods proposed in this research can be used to build such a predictive model.

3.1.3 Modeling of approximate patterns in software development process

Design patterns and anti-patterns play an important role in the practice of software development. These widely known notions have been mentioned in numerous publications, both scientific and popular. In most cases, their definitions are vague and rather conceptual than formal. Any attempt to put them in a strict mathematical model carries the risk of loosing important conceptual abstraction.

Rough formalism for descriptions of design (anti-)patterns In this research we propose a method for describing a design pattern in both formalized and vague way at the same time. The idea is embodied in the concept of a *rough software pattern* described in Section 6.3.2.

3.1.4 Automated support for code review process

Code review is an activity in the software development process, in which a dedicated person validates pieces of the system source code implemented by someone else. Research on the code review process shows that only as little as approximately two-thirds of the flaws can be spotted and reported by a human reviewer ([360]). Automated artificial intelligence algorithm can identify some design flaws and bad structures in the source code more efficiently ([367], [255]). Moreover, if such an algorithm is capable of predicting future potential flaws in the source code, its output can provide valuable information for the reviewer. The idea of detecting temporal patterns related to the evolution of bad structures in the source code can be used to detect such flaws or to improve the output of existing methods.

Support for code review activities We believe that this research can provide the foundation for a tool, designed for a human-expert reviewers, that enables them to focus especially on areas in the source code where bad structures are likely to appear. We believe a tool of this kind would improve a flaw detection ratio of such an expert.

3.2 Hypotheses

H1: There are statistically significant temporal patterns in software development process that can be used to predict the appearance of anti-patterns

Anti-patterns and bad code smells appear in developed software systems quite frequently, even though they are widely described in both scientific and popular literature. Probably, there are situations, which can be observed early in the software development process, that are early indicators for the future appearance of anti-patterns in the software structure. Hypothesis H1 states that such situations can be identified and their appearance is a good discriminator of whether the evolution of the fragment of the source code is drifting towards bad or good structure.

H2: Incorporation of expert knowledge can produce more efficient data mining algorithms in the domain of software development process

Hypothesis H2 is related to H1. In a sense, it can be perceived as its specific extension. It states that incorporation of expert knowledge in the domain of software engineering (in the typical mining algorithms) can produce a more efficient method of discovery of spatio-temporal rules in the software evolution.

3.3 Research goals

G1: Formal model of design (anti-)patterns, code smells, and their evolution

Goal G1 is to introduce a formalism that enables one to formally represent design anti-patterns, code smells and their evolution. We assume that such a model must be founded on graph theory (see Section 2.2.11).

G2: Approximate model to represent static patterns in software systems

Since software design (anti-)patterns are vague concepts, often without any strict definition, it is natural to use a formal model with immanent vagueness. Goal G2 is to define a formal approximate model for the description of spatial patterns in software design. This is strictly related to goal G1.

G3: Model to represent temporal patterns in software development process

Goal G3 is similar to G2, but positioned in the domain of temporal evolution patterns. G3 is to create a formal model that represents temporal relations between different evolving structures in the software source code. Clearly G2 and G3 together are related to a formal model for representing spatio-temporal relations in the software source code.

G4: Efficient mining algorithm especially fitted to the domain of software development process

Goal G4 is to create an efficient mining algorithm that would allow for localizing both structural and temporal patterns in the software development process. The algorithm should have the following properties:

- It should be capable of using an approximate description of structural and temporal patterns, as described in goals G2 and G3,
- it should be specifically fitted to the domain of software development process. This specialization must allow for good quality of detection or classification, while reducing the computation cost, as described in Hypothesis H1,
- it should be adaptive, that is, it should be capable of identifying new spatio-temporal rules during the evolution of the system, without the need for massive computation.

Chapter 4

Data mining fundamentals

This chapter defines some fundamental concepts of data analysis techniques that are either used or referred to in this research. Their comprehension is necessary to understand Chapter 6, which presents the model and the algorithms that are used to extract knowledge about the software development process. Readers familiar with topics related to machine learning and data mining may skip directly to chapter resume in Section 4.4.

4.1 Machine learning

Machine learning ([73]) is a sub-domain of artificial intelligence, which focuses on computer algorithms that learn from data.

Typically, when a computer is intended to perform a certain task, the programmer must implement a dedicated program. It is the programmer who should know how the task is to be performed and they should explicitly formulate it in the form of an algorithm. This is not the case in the domain of machine learning algorithms. Here, it is the algorithm itself which, in the process of learning from data, finds a proper way to perform the task. A rough classification of machine learning algorithms and examples of tasks they perform are given in the remainder of this section.

4.1.1 Supervised and unsupervised learning

We might assume that the goal of machine learning is to find an unknown function F that for any input x must provide a valid output y (see [282]).

In *supervised machine learning*, the algorithm learns from *training data* - a set of previously known examples of valid (x, y) pairs, such that $F(x) = y$ for all x. Given such input, the algorithm must find a representation of F that best matches the examples from the training set. We will denote this representation F'. Ideally, $F' = F$.

A good example of supervised learning is hand-written digit recognition. In such a setting, the algorithm is usually given a dictionary of labeled images, where the image contains a scanned hand-written digit and the label with the digit. Clearly, the image is the x and the label is the y in our example.

Test data is another set of examples which are used to verify how well F' resembles F. Once the algorithm has been trained on the training data and F' is defined, we check for how many x-es from the test data, $F'(x) = F(x)$ holds and for how many it does not. Clearly, the higher the first number and the lower the second number, the better the training process is[1].

The difference between supervised and *unsupervised machine learning* ([146]) is that in the latter case the training set is not available and the algorithm has to detect the structure of data without it. A typical example of unsupervised learning task is the *clustering* problem, where the algorithm has to partition a large set of objects into several clusters in such a way that the objects in one cluster are similar to each other, whereas the objects in two separate clusters are not. Here, the aforementioned function F is defined on the set of all input objects and has values in the set of clusters. Therefore, any pair $(x, y) \in F$ denotes the fact that object x belongs to cluster y.

Clustering is widely applied in market segmentation strategies, where customers are divided into several clusters, on the basis of their behavior. Data about all the activities of customers (such as purchases) is collected and then a clustering algorithm is run on it so that the customers are divided into distinct groups, where - slightly simplifying - each group represents a set of customers with similar market habits and any two customers from different groups behave differently. With such knowledge, companies can work on specific sales and marketing strategies tailored for each group.

4.1.2 Data mining

Data mining is a field of computer science whose main goal is to extract knowledge from massive data sets. It makes use of different methods and

[1]For more detailed discussion on the quality assessment please refer to section 4.1.4.

techniques, such as statistical analysis or supervised and unsupervised machine learning. The practical motivation for using data mining techniques is to detect initially unknown *patterns* in the data and then use them to infer knowledge about the nature of phenomena described by the data. In the end, the knowledge is typically used in decision support systems.

Predictive and descriptive tasks

Data mining is a very broad and vague concept, but the contexts in which data mining techniques are used can generally be divided into a set of *predictive tasks* and *descriptive tasks*. In the former case, the algorithm extracts knowledge from a set of known data and uses it to predict future or unknown facts about the same or related data. In the latter case, the algorithm is used to extract knowledge, which is not explicitly used for prediction, but allows one to understand the source data better.

A good example of a predictive task is classification described in Section 4.1.3. In this case, the data mining algorithm extracts a predictive model from a set of labeled exemplary objects (*training set*) and then the model is used to find an appropriate label for unlabeled objects of the same type. The aforementioned example of hand-written digit recognition is a valid example of a classification task. Other examples of such algorithms include regression, time series analysis or value prediction.

Out of a variety of predictive data mining methods, this research mostly uses the classification algorithms on specially prepared data. For details, please refer to section 6.4 and to appendix A.

Examples of descriptive tasks include summarizing, clustering, sequence discovery or association rule mining. The following paragraphs provide some intuition and examples of the last two, as they are specifically referred to in this research.

Sequence discovery

Sequence discovery is a category of temporal pattern mining tasks. In this setting, the data mining algorithm has to find frequently occurring ordered sequences in a stream of time-indexed events. For example, these methods are used to find typical behavioral patterns of users who navigate through web portals. This application allows us to build e.g. a recommendation system based on these patterns ([354]). A typical Internet user clicks through many articles on a visited website. Since there are usually thousands of articles available on such a portal and only a small portion of them can be offered to the user, the server has to decide which should be selected. When we observe a pattern that within a single session the majority of users who read article A, B and C subsequently, usually navigate to article D (while having other options), then probably D should be offered to anyone, who has already navigated through the "A,B,C" path. Just like sequence discovery algorithms enable us to understand the behavior of Internet portal users, so they can help us to understand behavioral patterns of developers, who take part in the software development process. This analogy is the basis for possible applications of this research described in Section 7.2.

Association rule mining

Association rule mining is a similar task to sequential data mining, but it operates in the context of non-temporal data. Simplifying it a bit, it finds some regular dependencies between different properties from large sets of data describing complex objects ([27]). If we take the database of all car models produced in the last five years, and compare their properties such as size, weight, engine displacement and fuel consumption, then we can probably conclude that large and heavy cars with large engines have high fuel consumption. This is an example of an association rule in the database: the property "fuel consumption" is associated with a set of properties {"size", "weight", "displacement"}. The goal of association rule mining task is to automatically find such associations in given data sets. A formal definition of association rules is given in section 4.1.4.

4.1.3 Classification problem

Classification is one of the most common tasks in data mining. The goal for the algorithm is to assign (classify) any given object to one out of, usually, a few known categories. Typically, the task falls under the supervised learning scheme, since in most cases a training set with previously defined classifications is given. The following paragraphs provide a formalism for the classification problem which will be used in the following chapters of this thesis.

Suppose that we have a finite, non-empty set of *objects* U, each described by the same finite and non-empty set of *attributes* A. The values of attributes may be of different kinds, including boolean, numeric, symbolic and textual. Formally, each attribute is a function from U to its respective domain. The pair $S = (U, A)$ will be called *information system* and U will be called its *universe*. Let $a_1, a_2, \ldots a_k$, be all elements of A, so that $A = \{a_1, a_2, \ldots, a_k\}$. For any given $x \in U$ we can calculate vector A(x) $= (a_1(x), a_2(x), \ldots, a_k(x))$. We will call this vector the *feature vector* of x and denote it as $A(x)$.

The simple constructs of an information system is sufficient to describe (directly or indirectly) a variety of real-world data that is analyzed by computer systems. The aforementioned example of a set of car models can be perfectly fitted to this formalism, where U is the set of all models and “size”, “weight“, “engine displacement” are elements of A. Certain subsets of U may represent *concepts*, for example “a well-designed car”, “a safe car” or “an economical car”. Intuitively, the problem of *classification* is to tell if a given object from U belongs to a certain concept or not. The following paragraphs describe it in greater detail.

Formally, any subset $C \subseteq U$ can represent a concept. If we extend the set of attributes by adding a new element d, called *decision attribute*, we may use it to indicate if a given element of U belongs to C, by putting: $d(x) = x \in C$. More generally, U can be partitioned into multiple concepts $\{C_p\}_{p \in P}$, and then d is given as $d : U \to P$ and for a given $x \in U$ it indicates which concept x belongs to. Such an extended structure $D = (U, A, d)$ is called *decision table* and each concept will be called *class* or *decision class*. In a decision table the attributes from A are called *conditional attributes*.

In this context the problem of classification is to provide such a model that for any x it can compute the value $d(x)$, based on $A(x)$. In practice it falls under the pattern of supervised learning, where d is given only on a (small) training set $T \subset U$ and the algorithm (called here *classifier*) has to find a valid method to compute d on elements out of T.

In the famous data set *Iris* ([118]), U is a set of samples of three different iris flower species (*Iris setosa*, *Iris virginica* and *Iris versicolor*), A represents the measured length and width of petal and sepal and d indicates which species the given sample belongs to. The goal of a classifier is to tell the species of the flower, based on the sizes of petal and sepal.

Exemplary classifier Irrespective of the method used to construct it, a possible classifier can be expressed in the following sentences: “If the petal width is below 0.6, then it is *Iris setosa*. Otherwise, if the petal width is below

1.7 and the petal length is below 4.9, then it is *Iris versicolor*. Otherwise, it is *Iris virginica*". This intuitive description represents *decision rules* which are described in greater detail in the following section.

4.1.4 Patterns and rules

Let $DT = (U, \{a_1, \ldots, a_k\}, d\}$ be a decision table. A *rule* can be written in the form of the following formula : $\bigwedge_i a_i(x) \in D_i \Rightarrow d(x) = v$, or, with a free variable x omitted for the sake of simplicity, as $\bigwedge_i a_i \in D_i \Rightarrow d = v$. We will denote the antecedent of it as ϕ and the consequent ψ and use a shortened notation $\phi \Rightarrow \psi$. Each "$a_i \in D_i$" clause in ϕ which puts a constraint on a single attribute, will be called *atomic condition*.

In the context of DT, for a given x from U, values of attributes (including d) are known, and therefore the formula can be interpreted and rewritten without free variables, so that it becomes a logical sentence. Depending on its boolean value, we can introduce two notions:

Definition 6. *A given object $x \in U$ is* matched *by rule $R = (\phi \Rightarrow \psi)$ iff ϕ is satisfied for x. The set of elements matched by R will be denoted $Match(R)$.*

Definition 7. *A given object x* supports *$R = (\phi \Rightarrow \psi)$ iff ϕ and ψ are both satisfied for x.*

Basing on the two notions, we can derive some elementary quality measures for the rules:

Definition 8. *Let s be the cardinality of elements in U that support R. We will call s an* absolute support *of R. A* support *of rule R, denoted by $supp(R)$ is a fraction of elements that support R in U, given by formula: $supp(R) = \frac{s}{|U|}$.*

Definition 9. *Let the cardinality of a set of elements from $x \in U$ that are matched R be denoted by r. Let the cardinality of a set of elements from $x \in U$ that support R be denoted by s. A* confidence *of rule R denoted by $conf(R)$ is given by formula $conf(R) = \frac{s}{r}$.*

Intuitively, the support measures how well a given rule covers the universe U and the confidence measures how accurately it describes data in the decision table. Ideally, we want a rule to have both high confidence and support, but usually in a particular application, we have to decide if we can sacrifice one of these characteristics in favor of the other.

Rule-based classifier

We will say that two rules $R_1 = \phi_1 \Rightarrow \psi_1$ and $R_2 = \phi_2 \Rightarrow \psi_2$ are *conflicting* on some element $x \in U$, if x is matched by both rules and $\psi_1 \neq \psi_2$.

A single rule typically matches a small subset of U, a few rules together can match a larger part of it. For a given set of rules $RS = \{R_i\}_i$ we define the set matched by them as $Match(RS) = \bigcup_i Match(R)$. Any set of rules that matches complete U can be used as a classifier, as long as the rules in RS are not conflicting on any element of U. Indeed: for any element $x \in U$ we can find a rule $R = \bigwedge_i a_i \in D_i \Rightarrow d = v$ that recognizes it and then puts v as a classification output for x. We will call such a classifier a *rule-based classifier*.

Many approaches can be taken to tackle the problem of rule conflicts in a rule-based classifier. The simplest one is to ensure that the rules from RS do not conflict with each other. This can be done by building a *decision tree*, briefly described in the following paragraph. Other approaches include voting ([59]), ordering the set of rules with respect to some rule quality metrics ([163]), naive Bayes approaches, where we try to find the most probable output of the conflicting ones ([74]), or other, more complex methods (e.g. [211], [47]). This aspect is beyond the scope of this thesis, which just incorporates existing algorithms to build conflict-free rule-based classifiers.

Decision tree Decision tree is a special form of representing a rule-based classifier. We may depict it as a binary tree where each internal node contains one atomic condition and the leaves contain decisions. Then every path from root to a leaf forms a single rule, where ϕ is built from conditions from internal leaves and ψ is defined by the value from the leaf. The conditions are combined by conjunction, according to the following principle: For a given node n that contains a condition $cond = a_i \in D_i$, if a path connects n with its left child, then take $cond$, otherwise take $\neg cond$. The diagram 4.1 shows an exemplary decision tree.

Human readability There are many different methods and algorithms for building a classifier from data, including neural networks ([145]), support vector machines ([82]) or Bayesian networks ([97]). However, rule-based classifiers have one important property: The model they produce can be used directly to represent knowledge about data in the form that is immediately comprehensible for people who do not understand the domain of machine

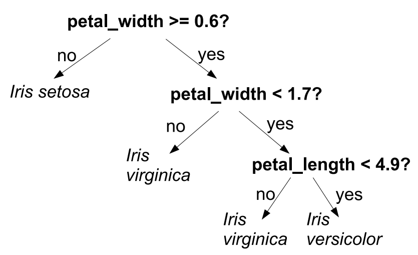

Figure 4.1: Exemplary decision tree for the classification of the Iris dataset.

learning. If we take the exemplary classifier for the Iris data set from the beginning of this section 4.1.3, or an equivalent decision tree from figure 4.1, they are apparent for anyone who understands how a plant is structured. Such a person does not need to know what a neuron, kernel function or probability distribution is. Consequently, we can apply machine learning methods to various domains and treat them not only as tools that help to find certain patterns, but also as a means to discover domain-specific knowledge.

This research follows the same paradigm: we use rule-based classifiers to find spatio-temporal patterns in the software development process and express these patters directly in terms of the software engineering domain.

Classifier quality measures

Frequently, when the classification algorithm is applied to some real world data, the domain of the decision attribute d is a boolean function that is the characteristic function of a concept. In this context, the classifier is used to describe the concept in terms of values of attributes from A. We will call such a classifier the *boolean classifier*. Please note that if the co-domain of d is a finite set, then the general classification problem can easily be decomposed into a series of boolean classification problems. Indeed: For each $v_i \in codomain(d)$, we can build a classifier that recognizes a concept $\{x \in U : d(x) = v_i\}$.

Suppose that we want to build a classifier c, which resembles function d, whose definition is unknown, but we know its values on a training set. Ideally, we would like c to be equal to d, but in reality this is hard to achieve, and we only want to know how "far" c is from d. In order to formalize it, we need to introduce some notions and quality measures applicable to a boolean classifier:

Definition 10. *Let $DT = (U, A, d)$ be a decision table, where $d : U \to \{0, 1\}$ is a binary function. Let $c : U \to \{0, 1\}$ be a classifier for DT. We will say that a function $f : U \to \{0, 1\}$ accepts x iff $f(x) = 1$. Otherwise, we will say that f rejects x. In the context of c:*

- True Positive *is such an element $x \in U$ that both d and c accept it. A set of all true positives will be denoted TP.*
- True Negative *is such an element $x \in U$ that both d and c reject it. A set of all true negatives will be denoted TN.*
- False Positive *is such an element $x \in U$ that d rejects it and c accepts it. A set of all false positives will be denoted by FP.*
- False Negative *is such an element $x \in U$ that d accepts it and c rejects it. A set of all false negatives will be denoted by FN.*

If the classifier is not clear from context, the above notations will be tagged with a subscript. For example TP_c stands for a set of true positives for classifier c.

Intuitively, the true positives, are those elements, which are properly accepted by c. True negatives are those which are properly rejected by it. False positives and negatives are elements that are wrongly recognized by c, respectively: accepted or rejected. Ideally, we would like TP and FP to cover complete U and FP and FN to be empty. In practice, we want FP and FN as small as possible. This can be quantified:

Definition 11.

- True Positive Rate*, denoted by TPR, is given by $TPR = \frac{|TP|}{|TP|+|FN|}$.*
- True Negative Rate*, denoted by TNR, is given by $TNR = \frac{|TN|}{|FP|+|TN|}$.*
- False Positive Rate*, denoted by FPR, is given by $FPR = \frac{|FP|}{|FP|+|TN|}$.*
- False Negative Rate*, denoted by FNR, is given by $FNR = \frac{|FN|}{|TP|+|FN|}$.*

These elementary measures can be used to assess the quality of a binary classifier. Clearly, we want the sum $TPR + TNR$ to be close to 1 and both TNR and TPR - to be close to 0.

Please note that usually a single factor alone is not sufficient to measure the quality of a classifier. Indeed, if we consider a naive classifier $c(x) = 1$, then it has high (read: proper) TPR and low (again: proper) FNR values. Therefore, we either have to look at all four factors together or we can derive more generic quality measures from them:

Definition 12 (classification precision and recall)**.**

- Precision*, denoted by* $precision(c)$*, is given by* $\frac{|TP|}{|TP|+|FP|}$.
- Recall*, denoted by* $recall(c)$*, is given by* $\frac{|TP|}{|TP|+|FN|}$.
- Accuracy*, denoted by* $accuracy(c)$*, is given by* $\frac{|TP|+|TN|}{|U|}$.
- F-measure *(alternatively* F1-score*), denoted by* $F-measure(c)$ *or* $F1(c)$*, is given by* $\frac{2\times|TP|}{2\times|TP|+|TN|+|FN|}$.

Please note that F-measure is actually a harmonic mean of precision and recall, and accuracy is actually a fraction of correctly classified elements.

Intuitively, recall measures how well the classifier is able to identify all the instances it should accept. Precision measures how well the classifier distinguishes accepted and rejected instances. Both accuracy and F-measure are generic measures that take all factors into account and show how well the classifier behaves in both accepting and rejecting instances. In general, however, F1-score is less biased by uneven distribution of decision classes.

Quality measures in information retrieval In the above context we measure the quality of the classifier by checking how well it identifies individual given instances one by one. We may also think of another situation in which we want the model to return all accepted instances from a large set at once. Conceptually it can be viewed at as if we want the classifier to tell in advance all possible instances it would accept. This is a specific context of *information retrieval* and the objects given in such a way are called *retrieved objects* and are denoted by $Retr$. The objects that should be accepted are called *relevant objects* and are denoted by Rel. If we know these two sets, we can compute quality measures such as precision and recall in the following way:

Definition 13 (information retrieval precision and recall)**.**

- Precision *is given by* $\frac{|Retr \cap Rel|}{|Retr|}$.
- Recall *is given by* $\frac{|Retr \cap Rel|}{|Rel|}$.

Quality measures for non-binary classifiers If we take a multi-class decision table DT with a decision table d and a corresponding classifier c, each row x such that $c(x) = d(x)$ is clearly a properly classified x. We can treat it as a true positive. In the context of precision we can consider all other rows to be false positives. This allows us to compute precision and recall according to the above formulas and consider F1 to be their harmonic mean. Since accuracy is a fraction of properly classified rows, and we know the number of such rows, we can easily compute accuracy accordingly. Measures computed in this way are called *micro-averaged-measures* or *μ-measures* (e.g. μ-accuracy) (see [312]).

In fact, in micro-averaging we only look at the number of true positives (i.e. properly classified instances) in the multi-class classification. The ideas of false negatives and false positives cannot be universally defined for all contexts. To cope with that problem we may reduce the multi-class problems to binary sub-problems: Please recall that a multi-class classifier can be decomposed into a set of binary classifiers if the number of decision classes is finite. However, the above formulas for quality measures do not apply directly. In this case we need a more sophisticated definition:

Given a decision table DT with decision attribute d and decision classes $\{d_1, \ldots, d_n\}$, as well as a corresponding multi-class classifier c, we can take each class d_i individually and construct a binary decision table DT^i and a binary classifier c^iaccording to the following rules:

- DT^i and DT have the same set of conditional attributes, and the set of objects.
- DT^i has decision column d^i defined by:

$$d^i(x) = \begin{cases} 1 & \text{if } d(x) = d_i \\ 0 & \text{otherwise.} \end{cases} \tag{4.1}$$

- c^i is given by:

$$c^i(x) = \begin{cases} 1 & \text{if } c(x) = d_i \\ 0 & \text{otherwise.} \end{cases} \quad (4.2)$$

Conceptually, such a classifier assigns only one decision class d_i and separates it from all other original decision classes. Since c^i and DT^i are both binary, we can compute the aforementioned quality measures for each d_i individually. We can then aggregate the results to produce corresponding quality measures for the original multi-class problem (see [312]): A *macro-averaged* measure is simply an arithmetic mean of this measure computed for all individual d_i. For example *macro-averaged-F1-score* is an arithmetic mean of all F1-scores computed according to the above construct. The same applies to precision and recall.

The macro-averaged measures can be biased if the decision classes are not distributed evenly in a set of objects. To cope with that problem we may use a weighted average in place of an arithmetic one, and weigh individual measures by the number of objects with corresponding decision. More specifically, the weight of a measure computed from DT^i and c^i is the number of rows x from DT, such that $d(x) = d_i$. We will call such measures *weighted measures* (e.g. *weighted precision*).

4.1.5 Overfitting and pruning

When a machine learning algorithm is building a classifier on a given training set, it takes available data examples and tries to find some patterns in the attribute values that are related to the value of the decision attribute. In this setting, we hope that the available training data is representative and any knowledge derived from it can be directly generalized to a broader context. Unfortunately, this is not always the case. If the data in the training set is too specific, then there is a risk of *overfitting*. This is a situation, in which the classifier has very good quality measures on the training set, but it misclassifies many objects located outside.

For example, let us consider that we want to build a classifier that recognizes a dog breed on the basis of coat color. A rule that says "If the object has 86 black spots, then it is a Dalmatian" is clearly overfitted. We know it because we have some domain knowledge of the types of dog coats and breeds and we can spot that the rule is too specific. It would probably help us to modify or remove this rule from the classifier, as one that can decrease its predictive quality. One of the possible modifications would be to change its antecedent to “the object has more than 50 black spots” or even “The object has a large number of black spots“.

The above example clearly shows that the domain expert knowledge can help us to cope with the problem of classification overfitting. A survey of similar approaches is given in [357]. Also in this thesis we try to incorporate the software engineering expert knowledge, together with the inherently approximate model, based on the concept of a rough set, to reduce the risk of overfitting and increase quality and applicability of extracted knowledge. Further details are given in the following sections of this chapter as well as in Section 6.3.2.

There are other methods that address the problem of overfitting (see [355]). In the case of rule-based classifiers, the methods typically fall under a single scheme: If the rule is too specific, we remove it from the classifier or we generalize it. Specificity of the rule can be expressed in terms of its quality metrics such as length, its support and confidence or *rule information measure* (see [362]). This process is called *pruning*. In the case of a classifier expressed by a decision tree, the technique is usually realized by removing some longer branches from the tree, so that paths from root to leaves are shorter (see [123]).

4.1.6 Discretization

Discretization is one of the techniques used to cope with the problem of data which is too precise. This often arises from accurate sensors that measure some actual quantity in the real world. For example it may be a thermometer which measures body temperature. Let us consider two rules that could model medical knowledge that would enable us to make a diagnosis: $R_1 =$ “When temperature is 38.5°C and there is a headache then the decision is influenza” and $R_2 =$ “When temperature is 38.75°C and there is a headache then the decision is influenza”. They both represent the same portion of medical knowledge, but they are based on different values provided for a given attribute. Intuitively, we would like to represent this knowledge in a single rule, similar to ”If temperature is high and there is a headache, the decision is influenza”. It is not important what the exact value of the temperature attribute is, it is sufficient to know if it is high or not. Formally, we would say that it is important if the value of an attribute falls into a certain subset of the attribute domain. This means that we can modify an information system by replacing the attribute with a new one that only determines which subset the value belongs to. In the example of body temperature presented above, we would replace the real-valued interval $[35°\mathrm{C}, 42°\mathrm{C}]$ into a small set of labels : $\{'very\ low','low','normal','above\ normal','high','very\ high'\}$, where each label corresponds to one of subsequent and adjacent intervals of [35,42] that partition it. The following definition formalizes the concept of discretization:

Definition 14. *Let $IS(U, A)$ be an information system and $a \in A$ be an attribute $a : U \to D$. Let D be partitioned into $\{D_i\}_{i \in I}$. Let function $disc : D \to I$ be defined as $disc(x) = i$ iff $a(x) \in D_i$.*

Let us define an attribute a_{disc}, given by $a_{disc}(x) = disc(a(x))$. We will say that a_{disc} is a discretization of a, and information system $(U, A \setminus \{a\} \cup \{a_{disc}\})$ will be called IS with discretized attribute a.

In data mining we discretize certain attributes, not only to simplify concepts in the data, but also because it may actually improve the quality of prediction models ([344]). Approaches to the problem of discretization can be categorized by a few aspects, such as the number of variables, the number of discretization stages, locality, autonomy and staticness (see [104], [128]). Some papers (e.g. [264], [347]) mention the use of domain knowledge in the process of data discretization. In this research we follow this approach and try to incorporate software engineering knowledge in the discretization of the respective data.

4.2 Approximation

Typically, machine learning methods are applied in order to model and reason about real-world phenomena. We can thus describe certain concepts in a formal model and determine certain dependencies between different concepts.

For example, if we take medical data gathered during patient examination, we think about concepts such as 'influenza', 'cold', 'fever', 'mild headache' or 'severe headache', etc.. Actually, what the doctor does, before making a diagnosis, is to find some associations between these concepts. Their expertise enables them to know that the concept 'influenza' is usually related to the concept of 'fever' and 'severe headache'.

Please note that all mentioned concepts are not crisp - they all contain some component of uncertainty or fuzziness. We will call them *vague* concepts.

In the aforementioned example, we cannot clearly tell where a 'fever' starts: If we agree to a certain formal boundary temperature, say 38°C, we must accept that 37.999999°C is not yet a fever, which is clearly counter-intuitive. Similarly, we cannot easily distinguish between mild and severe headache, and sometimes it might be difficult to distinguish between influenza and a cold.

The above example shows that in order to efficiently apply machine learning algorithms in the real-world, we have to cope with the problem of vagueness not only on the level of data and concepts, but also on the level of reasoning.

In this research some concepts are derived from the rough set theory, which is a formalism that allows us to incorporate vague concepts in data mining models.

4.2.1 Rough sets

A *Rough set* ([272]) is an extension of a classic set, which incorporates the concept of vagueness. In a classic set theory, for a given set X, and any element x, we can tell that either x belongs to X or it does not belong to it. This is not the case for a rough set: Here we can tell that one of three options is possible: x may definitely "belong" to it, it may not "belong" to it, or neither of the two preceding conditions is satisfied for sure. The last case may be interpreted as an uncertainty, whether x is a member of the set, or - in other words - that we are not certain if x belongs to it. Referring back to the previous example, we could say that 37.5°C does not "belong" to the set of fever temperatures, 38°C – does, and in case of 37.75°C – we cannot tell. [2]

The following paragraphs provide a formal definition of a rough set.

Definition 15. *A* rough set $R = (\underline{R}, \overline{R})$ *is a pair of classic sets such that* $\underline{R} \subseteq \overline{R}$. $\overline{R}$ *is called the* upper approximation *of* R, $\underline{R}$ *is called the* lower approximation *and* $\overline{R} \setminus \underline{R}$ *is called the* boundary region.

Intuitively, $\overline{R}$ is a set of the elements that may belong to the rough set, $\underline{R}$ is a set of elements that certainly belong to it, and the boundary region comprises those elements whose containment in R is not certain.

[2] Another intuitive example from the domain of software engineering was given in section 2.2.7

The purpose of using rough sets is to express vague concepts in the process of usually complex and large data analysis. Please recall that information systems and decision tables, defined in section 4.1.3, are a universal form of expressing such data. Rough sets can be defined in the context of an information system, so that they address problems with inconsistent data.

Definition 16. *Let $IS = (U, A)$ be an information system. The* indiscernibility relation *on U with respect to attributes subset $B \subseteq A$ is defined $IND_B = \{(x, y) : \forall a \in B\ a(x) = a(y)\}$. IND_A will be denoted by IND for the sake of simplicity.*

For any $B \subseteq A$ IND_B is an equivalence relation ([271]). Its equivalence classes will be called *indiscernibility classes*.

For any crisp set $X \subseteq U$, we will define the upper approximation of X as $upper(X) = \{x \in U : [x]_{IND} \cap X \neq \emptyset\}$. Similarly, we will denote the lower approximation of X as $lower(X) = \{x \in U : [x]_{IND} \subseteq X\}$. A rough set $R = (lower(X), upper(X))$ is an approximation of X. If we put an $X = \{(x \in U : d(x) = d_i\}$ for some d_i, then R is a rough-set-based approximation of decision class d_i.

Intuitively, the IND connects such a pair of objects that cannot be discerned from each other on the basis of the values of attributes from the information system. In other words, the information provided by these attributes is not sufficient to discern objects in the equivalence classes of IND. The definition of a rough-set-approximation of any set comes from the assumption that the lower and the upper approximation of any concept may only be built from available indiscernibility classes. We may look at the matter from a different perspective: The values of attributes are the only available description means. These means have certain limitations in their expressiveness: we are not able to describe concepts with granularity reaching deeper than the indiscernibility classes.

Please note that different sets of attributes from A define different partitions of U into indiscernibility classes. In the context of IS we may consider an expressive power of any subset $B \subseteq A$, by looking how different IND_B is from IND. This leads us to the following definitions:

Definition 17. *A subset $B \subseteq A$ is called a* reduct *iff $IND_B = IND$ and $\forall C \subset B$ $IND_B \neq IND$.*

The intersection of all reducts will be called a core.

Conceptually, a reduct defines the minimal set of terms of a language that is necessary to describe and discern all objects from a given information system without losing information. A core can be understood as the set of the most important elements of A.

In general, the problem of finding reducts for a given information system is NP-hard, but there are heuristic algorithms available (see [345]).

4.3 Sequential and temporal data

This section describes selected aspects of temporal data and the problem of mining temporal patterns. It contains a short discussion on selected problems related to acquisition of temporal data that are relevant for this research. You will also find formal definitions of different kinds of temporal patterns and the tasks of their mining.

4.3.1 Temporal data

Please recall from chapter 1 that temporal data is a representation of phenomena that can be ordered along time dimension or precedence relation. The types of temporal data can be categorized according to three concerns: acquisition, time representation and temporal ordering ([39]). The following paragraph briefly summarizes these aspects.

Temporal data acquisition

Temporal data can be acquired by observing a phenomenon over a certain period. Without loss of generality, we may assume that we observe a complex object. There are two possible ways of doing that: *event logging* and *data sampling*.

In the first case, we add a new item in the data, whenever a structure or property of an element of the complex object changes. We will call such a situation an *event*. Events are stored in a log, which is a sequence of log entries. Each log entry describes a single event, by providing the time when it was observed and a description of the event with all relevant information about changes in the structure or properties of the complex object.

In the case of sampling, a complete view of the complex object is periodically recorded (sampled) as a data item in the form of a *snapshot*. If there is no change in the object between two subsequent samples, then snapshots are identical. Also, if multiple changes have happened in between, we may not be able to reconstruct them from the snapshot, as one may be overwritten by the other. In particular, we can face the so called *A-B-A problem*, that is a situation in which the state of an object changes twice between two samples: first from A to B, then from B to A. In such case, the snapshots are identical, which may imply that there was no change, which is not true.

In this research we use the event logging approach, in which each event is bound to a single commit.

Time representation

Representation of time is an important aspect of temporal data. When time is continuous, then we can virtually assume that observable events are linearly ordered. In case of discrete time, this is not necessarily true. Additionally, we know the specific timing of an event only with some finite accuracy. In the domain of the analysis of the software development process, we usually assume discrete time, but the linear ordering of events is in fact guaranteed by the transactional nature of SCM system, which is the source of data.

Temporal ordering

In some cases temporal information available in the data can be limited only to the elementary precedence relation - either linear or partially ordered. This is not always a limitation. Sometimes, the timestamp information recorded in raw data provides only noise and it can be beneficial to remove or at least discretize it. This is the case when we focus on observation of one complex objects, whose behavior is dependent on some other object that is irrelevant for us. A network communication may be a good example: When we investigate the process of inter-systems communication, we are usually interested only in which order some messages are delivered to different software systems and we can probably forget about network latency in this communication.

In the proposed model, described in Chapter 6, we generally assume that precise timestamping of events is not important, and we only consider their temporal ordering. However, in some topics related to the locality properties of the software development process, described especially in Appendix A.2, the exact timing is necessary.

Allen's interval algebra

In [30] the author defines algebra and calculus over the set of intervals of $\Re$, which are by convention closed in their lower end and open in the upper end. The interpretation is that $\Re$ represents linear time and each interval corresponds to a continuous period of time. The relations defined in algebra represent temporal relations between the periods. For example interval $X = [x_1, x_2)$ *takes place before* $Y = [y_1, y_2)$ (equivalently: Y *takes place after* X) iff there is $s \in \Re$ such that $x_2 < s < y_1$. The algebra defines 13 different temporal relations between intervals, which comprise equality and 6 pairs of invertible relations. We will say that Allen's relation between X and Y defined above is *non-inverted* iff $x_1 < y_1 \vee (x_1 = y_1 \wedge x_2 < y_2)$. Conceptually, this means that X *takes place before* a non-degenerated sub-interval of Y. In other words, Y will last for some time after X has started. The non-inverted relations of these pairs are given in the following list:

1. X *takes place before* Y ($\exists s \in \Re : x_2 < s < y_1$)
2. X *meets* Y ($x_2 = y_1$)
3. X *overlaps* Y ($x_1 < y_1 < x_2 < y_2$)
4. X *starts* Y ($x_1 = y_1 \wedge x_2 < y_2$)
5. X *contains* Y ($x_1 < y_1 < y_2 < x_2$)
6. X *is finished by* Y ($x_1 < y_1 \wedge y_2 = x_2$)

Each Allen's operator A has its inversion A^{-1} (which is also Allen's operator) defined by: xAy iff $yA^{-1}x$.

Allen relations are used in this research as a formalism that allows us to express temporal relations in the software development process. However, instead of using intervals of $\Re$, we use intervals whose ends are represented by commits in the software evolution. Since the commits are linearly-ordered, all Allen's relation described above naturally apply to this context. For details, please refer to Section 6.5.5.

4.3.2 Time series

Time series is a special type of temporal data, in which data is a sequence of linearly ordered numbers (usually real-valued). A record of share price at a stock exchange is a good example of such data. The following definition formalizes this concept:

Definition 18. Time series *is a sequence of real numbers* $TS = (v_t)_t$.

The t index in the above definition is associated with time.
We can identify two reasons for which time series is such an important and widely researched method for the representation of temporal data. Firstly, this representation can be derived directly from real-world phenomena in many practical domains without any transformation and pre-processing. If we take a sensor that measures some real-valued quantity, such as temperature, vehicle velocity, heartbeat rate, etc., and sample its measurements every fixed time interval, we get a time series that is a valid record of the observed process. Secondly, time series is a very simple mathematical concept, to which we can apply a variety of well-known methods, including statistical analysis or calculus. The following paragraphs briefly summarize some of them.

Trend analysis and regression Sometimes the values present in a time series is the record of a process which is known or expected to behave according to a simple model, e.g. linearly. In such case we can use regression to reduce complexity of data and filter the noise. Even if the model is unknown, regression can be used to find it: If, after finding a regression model, we end up with a low error, then the output can probably be considered as trivial for analysis, yet a sufficient approximation of the initial data.

Trend analysis is a similar technique used in the time series analysis. In this task we want to detect such a model that adequately approximates a long term behavior of the time series that is not affected by local, short-term fluctuations. Also here simple linear models are frequently used. These models are commonly used to analyze e.g. financial data ([325]).

Periodicity detection Real-world phenomena can sometimes behave more or less periodically. If we sample some measures of such phenomena and store it in the form of a time series, it should resemble a periodic function. Simplifying it a bit, such a time series can be decomposed into a trend component and periodic fluctuation component. For example, if we take power consumption in households, we would expect that it periodically changes through the day in a similar way each day, because we would expect higher consumption in the morning, and in the afternoon, very low at night, and average through the day. We would rather not expect such a periodic daily fluctuation to be affected by long-term phenomena, such as citizen migrations, that can influence the total number of households using electricity. In such case, we could rather observe a long-term trend in the data.

In general, the task of *periodicity detection* in a given time series (v_t) is to find such two time series $(trend_t)$ and $(period_t)$ so that: 1) for any t, $v_t = trend_t + period_t$, 2) $trend_t$ is close (in terms of e.g. mean quadratic error) to a given regression function (usually linear) and $period_t$ is close in the same sense to some periodic function. The period of the aforementioned $period_t$ series can either be known in advance or it has to be identified as well in the course of finding such decomposition (see [109]).

Similarity measures for time series If we observe two different processes and each is recorded in the form of a time series of the same kind (the same value is measured), then we can ask the question how similar these processes are, and how the similarity can be mined from both time series. The problem is that one of the series, as compared to the other one, can be shifted, scaled or include some irrelevant trend. The similarity measure can be used to cluster the time series database so that the enormous dimension of data set can be reduced. This can enable us to efficiently reason about temporal data and use this knowledge in a prediction model. This general framework is frequently used to identify some temporal patterns in many areas, including, but not limited to, financial data analysis ([325]) or energy consumption models ([164]).

Time series in the software development process

In the software development process time series can be derived from software metrics: Indeed, if we measure and record the value of a metric at every revision (they cannot change between revisions), we get a time series. Analysis of such a time series, based on simple regression can be used as one of the predictors of phenomena in the source code development (see [274]).

4.4 Resume

In this chapter we have introduced some methods of knowledge discovery that are referred to in this thesis in the context of the software development process:

- We have explained the notion of supervised and unsupervised machine learning and provided exemplary algorithm types for both categories (see Section 4.1.1).
- We have explained what is a predictive and descriptive task in data mining (see Section 4.1.2).
- We have analyzed a few typical examples of data mining tasks, with special focus on classification and rule mining. On this basis we have introduced the concept of a rule-based classifier and explained its human-readability. The tasks were founded on data represented in information systems or decision tables (see Sections 4.1.3-4.1.4).
- The chapter discusses the problems of overfitting, data inconsistency, conflicting rules and vague concepts, which lead us to the notion of approximate modeling. The concept of a rough set is discussed as one of the approaches to this problem (see Sections 4.1.4-4.1.5).
- Special focus is put on the problems of mining temporal data: We have introduced the general problem of temporal patterns discovery and discussed some concepts related to modeling temporal phenomena. Specifically, Allen's interval algebra is briefly introduced (see Section 4.3).
- Certain quality measures for both rules and classifiers are explained in this chapter: starting from elementary ones, such as confidence and support, to more specialized, such as accuracy or F-measure (see Section 4.1.4).
- Last but not least, the chapter includes some relevant examples from the domain of software engineering, related to the concepts and notions explained above.

Chapter 5

Related work

This chapter discusses work related to this thesis. It is divided into two sections: Section 5.1 discusses bibliography on classic generic and domain-agnostic methods, whose elements are used in the present study. Their comprehension is necessary to understand the following chapters of this thesis. A reader familiar with techniques of statistical time series analysis, sequential data and process mining, as well as structure and graph mining can skip directly to section 5.2, which provides a thorough description of specific research into the domain of mining software repositories and the software development process.

5.1 Related generic work

This section describes work related to temporal pattern mining techniques. While the topic itself is very broad and exceeds by far the scope of our study, this section discusses certain concepts and algorithms that are explicitly or implicitly used in the present thesis or in any of related studies described in Section 5.2. The comprehension of these concepts is necessary to understand the following sections.

5.1.1 Process mining

Process mining is a data mining task in which we try to discover an initially unknown model of a process, by looking at many recorded process logs. The models can be expressed in terms of finite state automata ([80]), Petrinet ([300], [330]), workflow graph ([137]), event-driven process chains ([328], [234]) or languages typical for the description of business processes such as YAWL or BPMN ([329], [178], [58]). The techniques employed to discover the process model usually have a hierarchical structure, where the global model of the process is built from elementary casual relations between pairs of its steps. These relations can be constructed on the basis of a simple statistical analysis of the log. A good example of such an approach is given in [327]. On the basis of simple statistical analysis of available process logs, the α algorithm described in the paper builds a structure called *dependency-frequency tables* (D/F tables). This, in general terms, can be said to measure how often any pair of events (A, B) appear in the log, so that A directly or indirectly precedes B. Then, a few simple pre-defined heuristic rules are applied to reconstruct the *dependency-frequency graph* from D/F tables. Nodes of the graph represent event types and the directed edge (A, B) represents the fact that events A and B are likely to be in a causal relation. Finally, another set of pre-defined heuristic rules are applied to detect workflow constructs such as AND/OR-splits or loops. Similar algorithms are also presented in [96], [207], [231] or [340].

An alternative method is presented in e.g. [232] or [136] where a global, hierarchical representation of the process is adjusted in the course of the learning process. The advantage of such an approach is that it allows to capture non-trivial dependencies between distant steps in the process, such as non-free choices (i.e. a situation when a split in the process is determined by some past events). In [232], thanks to a genetic algorithm, the model is made to evolve so that it best fits to the known process logs. The general idea of [136] is built atop of a few similarity functions that are used to determine if two activities in the process can be abstracted to a single higher-level activity. In simple terms, the similarity functions measure how many spurious process traces can be generated if the two activities are merged together.

Further reading A good survey of various process mining techniques is given in [96] or [127].

5.1.2 Item-set mining and frequent episodes mining

The sequential log of a process is not always given in the form of a time series. Sometimes, rather than in numbers, data is given in terms of symbols from a non-ordered set. In such context, we can consider certain sub-sequences of symbols as patterns and apply mining algorithms to find them in a given process log. More generally, the pattern can be expressed as a set of partially ordered events. When the log is known up-front, then the adoption classic *a-priori* algorithm ([28], [29]) can be used to find sequential patterns. In [224] and [225], the authors use this algorithm to predict frequent episodes, defined as time-bounded occurrences of *events* ordered according to the given criteria, in the process understood as a sequence of events. The solution entails using the *sliding window* technique, briefly explained in the next paragraph. The study [141] introduces the *sequential pattern tree* - a data structure that provides efficient representation of sequential patterns and can be used to mine the ones that are frequent. All the aforementioned methods are based on the "anti-monotonicity" of frequent episodes, i.e. the evident fact that if a given sequence is frequent, then all its sub-sequences are also frequent and have equal or greater support. In [162] the authors describe a similar method for mining *complex frequent episodes*, which does not require the process to be linearly ordered.

Further reading A thorough summary of the methods applicable for mining frequent episodes can be found in [218]

5.1.3 Sliding window

Problems in understanding and mining temporal data may originate from the fact that sequences of data can be very large or even theoretically infinite, as in the case of *data streams*. *Sliding window* (see [341]) is one of the techniques used to reduce dimensionality of such data. Conceptually, it is intended to limit the analysis of a potentially very long sequence to a relatively small continuous interval, called window, and repeat the analysis for different locations of the interval. Usually, the window has a fixed length and the analysis is done for every possible starting point from the data. One can imagine that the window is *sliding* over the data from the first to the last item. Please note that, in such a setting, two subsequent windows usually differ by only a few elements, which may be convenient for various adaptive algorithms. The matter is conceptually shown in figure 5.1.3.

The sliding window can be useful in mining unbounded streams of very large data (e.g. [77]), in the presence of concept drift (e.g. [341]) or in hierarchical decomposition of the mining algorithm (e.g. [200]).

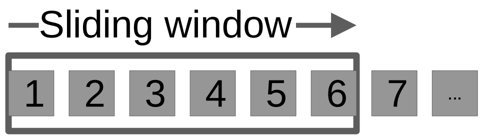

Figure 5.1: Conceptual presentation of the sliding window technique. At this particular moment, the window covers only the first 6 items of possibly infinite data stream. In the next step the window will cover items 2-7.

5.1.4 Structure and graph mining

Structure mining is a special task in knowledge discovery which aims at identifying spatial patterns in structural large scale data. In particular, *graph mining* focuses on detecting such patterns in a graph. The matter is thoroughly described in [25] and the following paragraphs just outline the most relevant aspects.

Graph isomorphism

Graph isomorphism is the problem of checking if two given graphs are isomorphic. Since graphs are an important means of modeling software structure, the matter has important practical significance in the domain of mining software repositories.

So far, it has not been established if the problem is solvable in polynomial time, nor if it is NP-complete. Thus, considering the current state of knowledge, it should certainly be considered computationally expensive. Yet, for relatively small graphs, the existing superpolynomial algorithms are practically efficient. Moreover, there are known polynomial time algorithms for certain categories of graphs, such as trees or planar graphs, and graphs with bounded node degree (see [183], [215] or [159]). This fact is particularly important for software structure mining, as real-life data in this domain usually yields graphs with reasonably bounded degrees (see 2.2.11). The paper [120] contains an interesting survey of existing exact-graph-matching algorithms.

Subgraph isomorphism

Subgraph isomorphism is the problem of verifying if the first-given graph (called *pattern*) is isomorphic with any subgraph of a second-given graph.

Despite similarities to the aforementioned graph isomorphism problem, it is known to be NP-complete ([81]). Here again, for certain types of graphs, such as certain types of sparse graphs, planar graphs with a fixed pattern or graphs with bounded node degree and a fixed pattern, the problem can be solved in polynomial or even linear time (see [253], [110]).

Induced subgraph isomorphism (see [76]) is a slight modification of the subgraph isomorphism problem, and its decision version is said to check if the first given graph (G_1 - *pattern*) is isomorphic to an induced subgraph of a second given graph G_2. Formally, this is true if there is a function f from G_1 nodes to G_2 nodes such that edge (x, y) exists in G_1 iff it exists in $G2$. Conceptually, the difference between the two problems is that in induced subgraph isomorphism, if an edge is absent in G_1, we require that the edge in G_2 between the corresponding nodes be also absent. In case of regular subgraph isomorphism, the edge may or may not be there.

There is a wide range of applications for the algorithms of the two aforementioned problems in the domain of software system analysis. It is used to find instances of design (anti-)patterns in a large structure of the source code. This approach is also used in the present study and is described in detail in Section 6.3.3.

Approximate and non-deterministic graph-matching

In practical applications, when applying mining algorithms to graph data (derived from e.g. software system structure), it might be sufficient to apply approximate heuristics to typical (sub-)graph isomorphism problems. Sometimes, it is even more desired to use such a method, because it can be helpful in reducing noise present in the original graph data. If a graph (or a subgraph) is approximately equal to another graph, we will say that they *match* in order to differentiate this notion from classical definition of isomorphism. You will find a short description of a few examples of graph matching algorithms in the following paragraphs.

Distance-based methods originate from string-matching algorithms. The notion of *editorial distance* between two textual variants can be generalized to graph data with the use of elementary graph change operations, such as *deletion*, *insertion* or *substitution* (for labeled graphs) of both nodes and edges. In some applications, more sophisticated changes, such as *splitting* and *merging*, are used as well (see [33]). A sequence of change operations applied subsequently one after another on a given graph will be called an *edit path*. Given the cost of each single change operation, we can define the cost of the edit path as the sum of costs of all operations included. The definition of the distance between two graphs G_1 and G_2 is straightforward: it is the minimal cost of a change path that transforms G_1 to G_2. Clearly, we consider G_1 and G_2 as matching if the distance between them is low. Algorithms for finding edit distance between two graphs are typically based on exploring the space of all possible mappings between nodes and edges, thus they are computationally expensive, which makes them applicable only for small graphs (see [66]). Some more computationally effective heuristic methods are based on the idea of optimizing local search instead of a global one, or of decomposing the problem of matching large graphs to a set of smaller ones. Both can be found in [257], [314] or [111].

Further reading A survey of edit-distance based methods for graph matching is given in [126].

5.1.5 Graph indexing

A special category of graph mining meta-techniques are based on the so called *graph indices*. They are particularly useful when pattern graphs need to be matched against a large database of graphs. Conceptually, the graph index plays a similar role as the index in a relational database: it allows to reduce search space to simple objects, such as vectors of real numbers, instead of more computationally-expensive graphs. Formally, let G be a set of graphs, function $Ind : G \rightarrow D$ is called *index* iff it satisfies the following property: For any $G_1, G_2 \in G$ if G_1 and G_2 are isomorphic then $Ind(G_1) = Ind(G_2)$. For practical reasons D must be efficiently filterable and in actual applications it usually is a vector space: $D = \Re^n$ for some natural n.

The matching algorithm is straightforward: Suppose that for a given pattern graph p we want to find $\{g \in G : p \text{ is isomorphic to } g\}$ and that $Ind(g)$ is known for all graphs in G. The search algorithm first computes $Ind(p)$ and then finds a subset $G_p = \{g \in G : Ind(g) = Ind(p)\}$. In the case operations on vectors of real numbers, such filtering can be done in logarithmic time with respect to the power of G. Moreover, if the Ind is well-suited for the particular domain, i.e. the probability of having two graphs in G with the same value of the index is low, then G_p should have substantially lower power compared to G. The last step is to perform a regular search for p in

G_p. Actual graph indices can be based on tree decomposition ([363]), paths enumeration ([131]), frequent subgraphs ([353]) or even algebraic properties of the adjacency or Laplacian matrices ([70], [216] or [297]). Some specific graph indices suited for the software development graphs described in Section 2.2.11, are also used in this study. For details, please refer to Section 6.3.3.

Further reading A variety of approaches to graph indexing is discussed in [25].

5.2 Related work in software development domain

This section describes in more detail specific studies in the domain of mining software repositories and the software development process. This has been a very popular subject of research for almost two decades, with significant dedicated conferences such as *International Conference on Mining Software Repositories*. The following sections describe a selection of available papers in the domain arranged into categories related to the present thesis.

5.2.1 Mining software repositories

Mining software repositories (MSR) is a special type of data mining task which analyzes rich data available in software SCMs, issue tracker, software documentation, mailing list archives and other data sources that are used during the software development process. There are different goals, models and algorithms used in the area of mining software repositories. In [175], the authors propose a taxonomy which allows one to arrange available research according to four aspects: type of data mined (code-named *what*), the purpose of the approach (*why*), the methodology (*how*) and the evaluation method (*quality*). Even though the taxonomy is based on papers published some years ago, the four-aspect classification still holds and with a slight amendment can be used to roughly categorize up-to-date research in this field (see [302]). The four aspects are roughly described in the following few paragraphs and related to the research presented in this thesis.

As regards the "*what*" aspect, the present thesis is mostly based on data concerning changes to software structure taken from the logs of SCM system and statically computed dependencies. Additionally, it is also partially based on defect statistics mined from the logs of the JIRA issue tracker[1]. Other sources of data mentioned in available papers include: SCM meta-data (e.g. [105], [117], [311]), bug and issue reports (e.g. [337], [191], [311]), developer's activity in Integrated Development Environment (e.g. [294], [219], [293]), external communication platforms such as e-mails or forums (e.g. [85], [44]).

The purpose of the thesis (the "*why*" aspect) is thoroughly described in Section 3.1. Generally, the goal is to be able to identify such temporal patterns in the software development process that are early indicators of deteriorating software quality. One can find a rich variety of other motivations for the MSR research, including, but not limited to:

- tools that improve the general comprehension of the software development process, such as various visualization techniques (see detailed bibliography in Sections 5.2.6 and 5.2.7) or specialized query languages (see Section 5.2.7),
- tools to identify the origin of certain patterns or defects in software structure (see Section 5.2.8),
- models that express temporal patterns (see Section 5.2.3) or temporal metrics (see Section 5.2.5),
- models for predicting complexity (see Section 5.2.3) or the number of defects in software (see Section 5.2.8),
- methods for mining non-obvious patterns in software structure (see Section 5.2.7).

Methods for mining software repositories used in the current thesis (the "*how*" aspect) are described in Chapter 6 and are summarized in Section 6.8. Other similar methods include approaches based on statistical analysis of data built from both static and temporal metrics (see Sections 5.2.8 and 5.2.5), various formal models (see Sections 5.2.3 and 5.2.7), numerous variations of pattern-mining techniques (see Sections 5.2.4 and 5.2.8).

The *quality* aspect of the MSR taxonomy is related to tools used to evaluate experimental output and is arguably the most fuzzy and unstructured concern. Methods used in MSR approaches include various types of expert assessment (e.g. [100], [147], [228], [243]), statistical evaluation (e.g. [55], [248],

[1]For details, please refer to Sections 2.2 and 6.1.

[32]), different methods for acquiring training and test data examples ([283], [191], [307], [290]). The present thesis describes a range of experiments and exploits all three types of evaluation methods (primarily the first one). For details, please refer to Appendices A and B.1.

In the sections below one can find a more detailed description of selected approaches in the field of mining software repositories, specifically related to the research described in the thesis. Again, similar or related studies are grouped together into separate categories and by convention at least one exemplary study is described more broadly in each category.

Synchronization of different sources of temporal data

Mining software repositories uses data from different temporal sources that, most commonly, are not synchronized. This appears to be a serious issue in the research on mining the software development process (see [42]). Therefore, the task of synchronization of different data sources is frequently taken as a part of data pre-processing in numerous MSR studies, and this part alone constitutes the core subject of many scientific articles.

In [42] the authors argue that significant information is lost when data sources are not synchronized automatically. Their research, based on experimental verification of Mozilla project, indicates that the recall of a heuristic approach for synchronization of SCM and issue tracker on this project is about 40% only.

Such observations have lead to the comprehensive introduction of tools that automate the process of synchronizing the logs from the two systems in software development. Probably the most popular (and widely used) method is the so called SCM hooks that does not allow to submit a commit to SCM if its message does not contain a reference to the corresponding issue in the issue tracker. Data used in the current thesis comes only from the systems with such a mechanism in place.

A deeper analysis of the same problem can be found in [56], where the authors have discussed the phenomenon of the *bug-feature bias* and *commit-feature bias* that were observed in data from open source projects. They are relevant for the problem of *defect origin* analysis described in Section 5.2.8. One of their major conclusions is that data is not evenly distributed over all types of bugs that actually appear in the development of software systems, which affects accuracy of known defect-prediction systems. Forcing developers to mark bug-fixing commits is suggested as one of the means to cope with the problem, yet the authors suggest that the data acquired in such a way still might be biased.

The problem of synchronizing logs obtained from different systems is also addressed in [177], [117], [366], [176], [42] or [311].

Data fetching and representation

All the systems that record a trace during the software development process constitute a very rich source of data that can be used in further mining. Therefore, one of the first decisions that a researcher must make is choosing the way to fetch data from these systems and selecting their representation to be used in the experiments. This section describes a few approaches to these two aspects and compares them with the solution used in the thesis.

Generally speaking, fetching and representing data can be categorized according to four aspects: type of data analysis, granularity, data model and time model, if applicable. These aspects are roughly described in the following paragraphs.

Dynamic vs static program analysis Though research in the field of mining software repositories can be based on various sources of data (see description at the beginning of this section), it is the program structure that is most commonly used. The methods of its reconstruction can be divided into two main categories: static and dynamic approaches (see [302]). In case of the former, the structure of the program is determined only by a static analysis of the source code e.g. during compilation. Variations of this technique are used in [166], [301], [43] or [315]. For further investigation into this topic please refer to [53]. Dynamic analysis, in which the structure of the program is reconstructed while the program is running, is used in [214], [57], [251] or [57]

This research relies on statically-build data created during dedicated program-source-code parsing. This decision comes from motivation described in Chapter 3: On the one hand, it allows the methods described in this study to be easily applied to real-life software development processes. On the other hand, this kind of method has been proven to be sufficiently accurate in the applications related to the thesis (see [129]). A detailed description can be found in section 6.2.

Data representation In [193] the authors yield a proposition of an universal XML representation of data from the software development process. The model includes information that can be directly taken from commit metadata, (e.g. author, time, etc.), information that can be taken from parsing modified files (e.g. source code structure, source code entities), information that can be precisely computed on the history for the modified entities, the number of added or deleted files, past co-change statistics (see Section 5.2.4) and information that can be computed heuristically (e.g. predicted defect distribution or bug-origin (Section 5.2.8), implicit dependencies between entities (Section 5.2.7). The proposed format structure of the XML file has not been widely accepted, yet the concepts proposed in this paper are still discussed and used in research to date (e.g. [217], [317], [348]). The most important ones are: 1) Universal, SCM and issue tracker-independent representation of the software development process log, 2) universal, programming language-agnostic representation of the software structure (various dependency graphs) and 3) capability of extending raw data with auxiliary information gathered during additional analyzing or mining raw data.

In this thesis, the exact XML structure recommended in [193] is not used, instead a model is implemented that perfectly adheres to the three characteristics outlined above. A detailed description of the model is found in Section 6.1.

Similar, universal models are also used in [222], [173], [190], [252], [134] [113], [105], [50], [90], [89], [88], or [177].

Evolution representation In the aspect of software evolution modeling - some research has also shown an attempt to provide a universal, technology- and process-agnostic representation of software evolution. Two common approaches appear in research papers: the ones that represent the evolution as a sequence of static snapshots (e.g. [134]) or as a sequence of changes (e.g. [249]).

Another important aspect of the software evolution model is its *temporal resolution*, that is, the minimal temporal distance of subsequent versions of the system source code that can be retrieved from the model. Here, approaches vary from very coarse models build on subsequent software releases (e.g. [198], [35]) through daily versions (e.g. [249], [252]) up to fine-grained tracking of elementary editorial changes done by the developers in their Integrated Development Environments (e.g. [293], [292]).

However, if the model is rich enough and captures the data with sufficient resolution, the difference between the above representations is just technical, since there is a function that translates one to the other. To some extent, this is also the case in the present study, since the model is based on a sequence of changes of particular files in the system source code, but it contains enough information to transform this to a sequence of snapshots or trace inter-commit changes for entities on any level of granularity down to single methods. For details, please refer to Sections 6.1-6.2.

Another discriminant of evolution models is the first-class entity, whose history is examined: it varies from complete systems (e.g. [190], [35] or [199]) through components, packages and files (e.g. [349], [91], [103]) down to single methods or classes (e.g. [130], [350], [1], [144]).

There are also other representations of software evolution that cannot be easily fitted into the above categorization. For example, in [65] the software development process is represented as a graph whose nodes correspond to commits and edges denote how much the connected commits have in common (in terms of the number of software entities changed in both commits). This specific structure can be built adaptively and a simple graph traversal algorithm allows one to calculate a few software development process metrics efficiently, but does not allow for a reconstruction of the process history. Other interesting approaches based on graph representation of software evolution are also presented in [49] or [138] where appropriately applied classic graph algorithms are used to respectively: identify unstable areas of software structure and find relations between system features and fragments in the source code that implement it. It appears that to-date there is no common, widely-adopted model to represent software evolution (see [229]).

5.2.2 Elementary commit statistics

One of the basic empirical methods for understanding the software development process, is a statistical analysis of the distribution of simple measurable properties of the commits, such as time, size, authors, etc. Such analysis allows one to streamline the following more sophisticated research in a more specific direction.

For example, in [240], the authors report that 10% of commits modify a single line of code, around 50% modify a maximum of 10 lines of the code and only as little as 5% modify more than 50 lines of code. A similar observation has been made based on experiments conducted for this thesis (see results in A.2). Additionally, similar statistics for relatively short time windows (up to 5 days) have been measured. It can be concluded that the software development process appears to be local in time and local in space (this topic is discussed in depth in Section 6.2.3). Therefore, an adaptive algorithm working on a stream of commits should be a more efficient way of analyzing software evolution data, as compared to an algorithm working on a stream of software snapshots.

Simple statistical information built directly from the SCM commits is also used in e.g. [238] and [31], where the authors try to evaluate developers' expertise in certain areas of the system on the basis of the distribution of the commits they made, or in [249], where the authors analyze the correlation between statistics concerning size of change and the number of defects in the respective source code.

Please note that statistics of this sort are actually very simple temporal metrics for the software development process. Research on the temporal metrics in general is reported in this chapter in Section 5.2.5.

5.2.3 Software evolution analysis

Much of the research effort is spent on understanding patterns in software evolution. This chapter describes important areas of evolution-related topics.

Human expert-aided software evolution analysis

Some research effort has been made to implement tools that extract evolution knowledge from parsed logs of the SCM or issue tracker systems and present it in an appropriate form so that it is subject to human expert evaluation and interpretation. In [326] the authors provide a tool that is founded on very similar concepts to those presented in this study: Apart from various dependency graphs, static and evolution metric, they also introduce the concept of *origin analysis*, which aims at determining history of a single software entity throughout system evolution, even if the entity has been relocated or renamed. The goal of the proposed approach is to provide a human expert with a set of cross-navigable charts and reports so that they can explore and understand software evolution. A similar approach is also described in [226], [94], [68] or [336] and a comprehensive survey of such methods can be found in [115].

Despite these similarities, there is a significant difference between the above and the approach taken in the present thesis. While the authors use the model solely to build the tools that visualize certain temporal metrics and patterns, this proposal goes further and the knowledge is an input for machine learning algorithms that extract new knowledge about the evolution. In other words, it is either the computer which interprets known facts and infers extra knowledge or this is done by a human expert. Delegating this task to the computer takes us one step further in efficient comprehension and monitoring of the software development process. One of the key conclusions of this thesis is that simple machine learning algorithms used on appropriately prepared data produce a useful model that can be used to predict system evolution in a fully automated manner.

Logical models

Temporal logic is a formal tool frequently used to model objects that change over time. The software development process is not an exception - some researchers try to build formal models of evolution and then use temporal logic to formally reason about it. In [134], the authors propose the use of a formal model to describe static patterns in object-oriented software and apply temporal logic to extend it so that it is also capable of describing temporal patterns in software evolution. The expressive power of the proposed language is shown on a few examples of temporal phenomena that can be useful in software engineering. A similar language, also built on the basis of temporal logic, can be found in [322] and [157]. What is common to all these papers is that the language defined there is not used to automatically reason about temporal patterns, but rather it is used as a tool for convenient description. In contrast to that, such automatic reasoning is described in [189], where the authors use temporal logic to automatically detect certain changes in the code. The changes are predefined in temporal logic languages and include typical refactoring operations, such as *"removal of a certain field in all classes that implement a certain interface"* . Similar approach is also used in [153]. The temporal aspect is modeled differently in the current study: In order to model temporal relations between phenomena observed in software evolution, instead of specific temporal logic (e.g. CTL), we use Allen operators (see Section 4.3.1).

Logical models are also used to reason about non-temporal properties of the software source code. For example in [140] or [243] we can see spatial pattern instances defined in a formal language. A more sophisticated problem is addressed in [315], where the authors define a formal logic language to describe the model of the source code structure and data flow and then build a static pattern detector based on it. The original solution was implemented with Prolog, but was then exchanged with dedicated SQL queries for performance reasons. The work is further extended in [338], where the formal method is used to detect structures in the software source code, which is not an entirely accurate instance of a given design pattern, but rather its approximation or variation. This method can be used to understand the structure of the source code that has been developed for a long time and its original constructs have been amended by unwanted editions. Similar problem is also addressed in [140].

Long-term evolution patterns

Research aimed at the problem of detecting temporal patterns in the software development process can focus on *long-term evolution patterns*, that is, phenomena that can be typically observed between subsequent releases, or short-term ones, which describe changes that happen between subsequent commits. This subsection provides a description of studies dealing with the first category, whereas the following section treats of the second.

The fundamental work [201] (later discussed in e.g. [203], [202]) gives a categorization of software systems with respect to their use and specification and discusses the way to understand evolution of each category by means of *Lehman's laws*. The most interesting category is *E-type* system defined as "*software solving a problem or addressing an application in the real world*", which is one that *Lehman laws* most obviously pertain to. The laws include: *The law of continuing change*, according to which the system must continuously adapt to changing environments and applications, *the law of increasing complexity* and *the law of continuous growth* according to which the system naturally and continuously increases its complexity and size and *the law of decreasing quality*, according to which the system's quality will naturally decrease along with its evolution. Clearly, the laws yield long-term evolution patterns, which have been the focus of research. Quite often research is based on simple statistical properties of software evolution. For example, in [69] the authors have empirically validated the laws on the basis of the evolution of popular open-source Integrated Development Environment. The conclusions supported only a few of Lehman's Laws (precisely: the laws of continuing change, self regulation and continuing growth), and, surprisingly, the law of declining quality was neither confirmed nor proven wrong. This is attributed to the specific nature of software, namely, to the fact that it is a community-developed open-source system. Such systems were also examined in [115], [305], [132] or [150] and all these papers have drawn similar conclusions.

A more sophisticated method for understanding evolution is given in [149], where the authors try to validate a model in which software evolution is approximated as a dynamical system with self-organized criticality. The consequence of such an assumption is that the evolution of such a system is determined by events that took place in the far past. By doing a statistical analysis of almost four thousand open source projects, the authors have proven the model wrong for long-term evolution patterns, but, interestingly, their results indicate that such models tend to be valid for short-term software development processes.

The matter of long-term evolution patterns is also described in [171], [40], [116], [98], [296], [334], [83], [124], [265], [72] [152] or [349] and detailed summary of research on this topic can be found in [151].

Short-term evolution patterns

Research on short-term evolution patterns can be approximately divided into the following categories: 1) patterns of a developer's editorial changes, 2) classification of changes done in a single or in a few subsequent commits, 3) identification of relations between different code modifications. The first category can be exemplified by [219], where the authors use a support vector machine model to predict if editorial changes done by a developer inside their Integrated Development Environment introduce new defects. The model is built from the simple attributes of a developer change, including their name, simple and cumulative software metrics, time of occurrence and others. One interesting conclusion is that such an approach is relatively effective (65-92% accuracy) when the analyzed history consists of up to approximately 250 past changes only, which in such context can be considered the threshold length of short-term evolution pattern. Other papers that fall into this category include [129], which focuses on identifying source code entities under heavy development or [156], which concerns typical elementary refactoring done by developers.

A good example of research from the second category is [155]. The authors build a statistical model based on the change of vector built from the values of various software metrics. The model introduces the concept of *metric fluctuation* and *change fluctuation* which are specifically aggregated values of metric changes evaluated during every commit. The changes are expressed in terms of entropy, a variety of distances, quartile deviation and quartile dispersion coefficient. This sophisticated model is eventually used to classify which software modules are most error-prone, more specifically, to indicate the top 20% of source code entities with the greatest number of defects. The model was empirically evaluated on a few open-source systems and the main conclusion is that introducing metric and change fluctuations improves classification accuracy by at least 20%. A similar concept, yet with simpler statistical model, was used in [276]. Other studies in this category are: [292], where the authors compare the method of identifying refactorings based on either editorial changes done in Integrated Development Environment or SCM logs, [289], where short-term evolution patterns are used to find defects density in software, or [191], where source code syntactical tokens are used to detect commits that introduce a defect into the system.

The third category can be exemplified by [366], which describes a method for identifying relations between different source code entities on the basis of how frequently they appear together in a single *transaction*. The transaction is here understood as a short-term pattern that consists of a few, usually subsequent, commits, which altogether form a structural change. Other studies in this category include [65], which describes the method of identifying a relation between different commits on the basis of the semantic change that they introduce.

Research described in this thesis is more affiliated to the second category: it analyzes how each commit changes the spatio-temporal relations in software structure. However, since these spatio-temporal relations may last for the entire evolution, in a sense, it also links to the first category.

5.2.4 Co-evolution analysis

Evolution patterns are usually bound to a single code entity or to a system as a whole. It means that we can treat two code entities as similar if their evolution has many common events (usually commits). In other words, it means that they change in a similar manner. Such a common change is called *co-change* and existence of many common co-changes is called *co-evolution*. If two different code entities co-evolve, then one can deduce that they are related or similar. Identification of such relations may play an important role in understanding global patterns in software evolution. This is, arguably, one of the most extensively studied paradigms in the area of mining software repositories.

In general, research might focus on different levels of granularity, starting from top-most elements, such as systems ([124]), down to methods ([366]). The method of identifying implicit dependencies on the basis of co-evolution patterns is described in [273]. In [51] the authors define simple co-change measures for classes and use it to empirically check what co-changing patterns are related to statically identified instances of design patterns in the code. In [371], the EROSE recommendation tool is evaluated. In principle it uses itemset mining technique on commit level to identify the lowest-level entities coupling. It doesn't however take into consideration sources of data other than SCM commits. The tool is part of Integrated Development Environment, so that whenever a developer changes a function in the code, the tool also suggests other places in the code that have previously been modified along with the one.

Author in ([365]) discusses the method of co-evolution analysis on the basis of *dynamic analysis* - the analysis of calls between system elements during its execution. The approach presented there contains an interesting concept of using historical data to generate initial set of co-evolving entities, so that dynamic analysis can be limited to a significantly smaller area.

In [124], Gall et al. propose a preliminary approach of using an item-set mining algorithm to identify logical coupling between large software entities, such as programs, on the basis of changes at release level. In [334], the authors define a special vector that contains timestamps of all commits that modified a given file. A quasi-euclidean[2] distance between such vectors represents information on how similarly the files were evolving over time. On the basis of such a metric, the authors run a clustering algorithm so that, in the end, we can identify groups of files that were evolving in a similar way. In this way, we can identify types or categories of file evolution. This input is then taken to a graphical algorithm that plots evolution categories on a diagram. We believe that this paper provides a small, yet significant step forward towards understanding co-evolution. In previously mentioned methods, research was more focused on identifying the co-change as such,

[2]Such vectors do not belong to a single euclidean space, so a special construct is needed to formally define the distance. For an exact definition of this metric please refer to the original paper.

here, however, we have a tool that potentially enables us to identify more fine-grained reasons for the presence of co-evolution and potentially see it as a temporal, ongoing process that needs further investigation.

5.2.5 Temporal metrics and evolution metrics

Source code metrics (see Section 2.2.7) are used to express a property of a certain fragment of the source code in the form of a real number. Similar concepts can be used to express properties of some temporal phenomena, such as the software development process.

Girba et al. in ([129]) provide a very simple metric that measures how the number of methods has been changing in a given class during system evolution. They use it to identify which are the most important parts of the system source code. Conclusions from their work include an experimentally validated statement that static metrics, which measure the size or complexity of source code entities, are not suitable to assess the importance of an entity. Temporal measures tend to outperform them in this category. A similar idea of measuring accumulated changes (usually with the use of *code churn* measure) on a software entity in order to approximate its importance in the system, its further evolution, its defect density, etc., is also present in [249], [285], [248], [246] or [179]. Related work constructed around temporal measures bases on developers' activity history or past defects, is described in [262] or [135]. Interestingly, also static measures can be used to predict future evolution of an entity and as such the approach is described in [332] and a combination of both static and temporal metrics can be found in e.g. [199].

In [290], the authors build a set of slightly more sophisticated temporal measures (e.g. the number of authors who modified the source file) in order to predict if a given software entity is to need refactoring soon. Experimental results show that the method is quite accurate (both precision and recall over 70%) in distinguishing source files that are not to be refactored. Another interesting statistical observation mentioned in this work is the fact that around 12% of commits in reference, open-source projects are refactoring changes. A similar approach can also be found in [35], [306] or [288].

Evolution metrics which are used to predict the number of defects in a software entity are also implicitly used in [276] and [275].

5.2.6 Visualization of evolution patterns

A very common approach to detect evolution patterns in the software development process is to use certain visualization techniques. The software development process is depicted on a graph, diagram, or a picture and certain software evolution patterns are depicted as specific shapes.

In [117], the authors propose one of the most elementary approaches, which is a simple graph that shows the growth of the system with respect to the number of source files added or modified. The graph is then used to discuss the growth ratio (linear or superlinear), or the proportion of files modified or added within a given period. In [265], the authors describe their tool *Herodotos*, which actually combines sophisticated techniques to detect certain static patterns, using simple graph generation, which can show e.g. the number of occurrences of a pattern over time. As suggested, a human expert can interpret such graphs to determine the correlation between different patterns or to understand the process of their co-occurrence. In [334], the authors propose an even more sophisticated visualization technique that allows one to detect the coupling between different files. It allows one to visually spot certain temporal patterns, such as periods of rapid development, large refactorings, or logical coupling between groups of files.

A similar method is also described in [51], where the authors use visualization of evolution patterns to identify co-changes in classes that are part of an instance of a design pattern.

A more advanced approach is presented in [198], where the author describes a visualization method that shows an evolution of a single class in the software source code as a diagram with subsequent rectangles, whose dimensions correspond to values of a source code metrics. In this paper, the author describes interesting temporal patterns in an evolution of a single class that can be observed in such a diagram. A situation in which a class appears in a version of the source code, and disappears from it shortly after is called a *day fly* pattern. In the diagram it is visible as a row with only a single rectangle in it. Similar graphical representations are given for patterns whose name is inspired by a far analogy to the evolution of stars in space, such as *supernova*, *white dwarf*, *red giant*, *stagnant* or *pulsar*.

What is common to all the aforementioned techniques based on visualization, is that the reasoning about the evolution of a software system or entity has to be done by a human expert who has to spot certain shapes or other indicators on graphs or gauges. Undoubtedly, while useful for software engineering, these methods are limited because the crucial part of the analysis and identification of certain patterns has to be done manually. One of the motivations for this research was to try to automate this process, so that some very good ideas proposed in the aforementioned papers can be taken as an input to an automated machine learning algorithm, so that we can build a tool that skips the step of visualization and graphical analysis and goes directly to conclusions about temporal patterns.

Visualization as a core method for understanding software structure and development is also used in [45], [79], [333], [270], [51] [99], [116], [346], [79] [326], [92] or [91] and a through summary of the topic can be found in [98].

5.2.7 Static code analysis

Static code analysis (SCA) is a method of analyzing software system only by observing its source code – without actually executing it. A general scheme for such a method is constructed as follows: At first the source code is parsed so that its structure is represented in abstract syntax tree and various *dependency graphs* (see Section 2.2.11). Such a representation allows one to derive further data about software, both quantitative (e.g. values of software metrics) and qualitative (e.g. instances of certain spatial patterns in these graphs). Elementary static code analysis tools are based on statically computed software metrics. For example [67] presents interesting research results on the method of identifying how readable the source code is. The authors build a Bayes network with variables denoting very simple, yet not always widely used, static software metrics (e.g. the number of commas being one of them) and validate the model against data collected in a survey conducted among actual developers. The study claims that meaningful identifiers have little impact on the readability of the source code, which seems contrary to the findings of classic publications [63] and [62]. This finding is related to one of the assumptions of the model described in this thesis, since we also assume that patterns are invariant to names of software entities. For details, please refer to Section 6.3.2.

A similar approach, where simple, statically computed software metrics are used to reason about practical source code features (such as readability), is also presented in [241], [228], [26] or [133], [228].

SCA for defect prediction

A related category of research papers is dedicated to the use of source-code-metrics-based static code analysis to predict defects in software. Such an approach, which started in nineteen seventies with works, such as [230] and [142], is still popular (e.g. [247], [236], [161], [43], [250]). A good summary of this type of research can be found in [246], [93], [320] and [174].

SCA for Design (anti-)patterns and smells detection

The area of application of the static code analysis most related to this research is its use in the detection of design patterns and anti-patterns. The following paragraphs present a few examples of such research. In depth summary of these methods can be found in [102], [114], [302], [24] or [321].

Elementary applications of static code analysis to detect simple mistakes and code smells based on analysis are described in [43], [336], [168], [241] or [243]. The general idea in these papers is always the same: First, the smells are formally described by a human expert in terms of dependencies between its constituents and their properties, usually expressed as values of certain software metrics. Next, the source code is translated into respective graphs (e.g. abstract syntax tree, dependency graph) and the values of a few software metrics are computed for code entities represented in the graph. Finally, an exhaustive search on such a structure is run in order to find occurrences of a given pattern.

The method proposed in [154] identifies design pattern instances in the source code in a two-fold process: first static analysis is used a find *design pattern candidates*, then dynamic analysis during program execution is used to validate if these candidates conform to the expected pattern behavior. It is worth mentioning that the dynamic analysis is still only used to detect static properties of the software source code structure (e.g. the fact that a certain software entity is invoking another one). The authors argue that neither approach alone (i.e. static and dynamic) can produce results of quality close to the one obtained by combining both of them. However, they also state that in this particular setting they were able to reach a very low number of false positives, but the number of true negatives was very large. It shows that the method might be too selective for real, noisy data, which is also explicitly confirmed by the authors, who admit that the goal was set to identify exact matches of the patterns and that the data about identification was given by a human expert.

The aforementioned pattern identification techniques are based on a strict definition of particular design patterns. However, they are, as already mentioned in Section 2.2.9, vague concepts of solutions and do not always fall upon a formal definition. Therefore the question arises if we can efficiently identify approximate patterns or structures similar to a given pattern. This problem is addressed in [315], where the authors make an attempt to identify a few variants of design patterns. The detection concept starts with a formally defined strict definition of a pattern called *canonical form*, which is described in the first-order logic with a language built with simple predicates that can be extracted from a statically built dependency graph. The graph is constructed during static code analysis and for some types of dependencies related to control flow, contains all theoretically possible edges. Moreover, the formal definitions of a few creational patterns are constructed in such a way that they are able to identify different variants of a given pattern. Altogether this yields a method that is capable of very efficient discovery of approximated design pattern instances. This method is also described in [338]. In the present thesis the problem of vagueness is addressed in a slightly different way: The formal description of a design pattern yields a crisp upper and lower approximation of it. For details, please refer to Sections 6.3.2 and 6.4.

Source code metrics are frequently used as the key input in design anti-pattern detection. In a typical approach code entities are described by a vector of values of corresponding metrics and statistical analysis or machine learning is used on such a table to identify any correlations (see [36]). Such an approach scales well, as we are able to gather large-scale training data labeled by experts ([204], [205]).

In [148] the authors describe the study of a *composite pattern* identification technique. The term relates to the structure in the software source code that contains overlapping instances of two or more different design patterns. The method is based on specific types of static metrics built atop simple metrics, such as the number of attributes or the number of methods, all computed on the basis of static code analysis. Conclusions include an observation that certain pairs of design patterns appear more frequently than others. Moreover, an interesting method of pruning *weakly overlapping* of pairs of patterns is described, which actually yields a proximity measure between instances of design patterns. A similar concept is also used in this thesis: the concept of *closeness*, and specifically *overlapping closeness*, described in Sections 6.5.1-6.5.2. The main difference is that in this study these notions are also defined in the entire evolution, rather than in a single static state of software.

Architecture mining

Architecture mining is a special type of structure mining, which is specific to the domain of software elements. Its goal is to detect unknown software architecture elements by looking at the source code. It is usually applied to legacy systems with poor or non-existing technical documentation and its goal is to understand the system design principles and high-level structure. It is mostly based on the methods of static code analysis described above. However, other methods of mining are used more broadly, and, what is more, the goal is usually to identify large-scale patterns, built of grand software entities, such as modules or packages. The summary of the matter and a categorization of various approaches can be found in [235] or [304]. The following paragraphs give a few examples of different types of methods used to mine system architecture.

The first category consists of simple graph analysis techniques, such as clustering or finding connected components that are applied to the usually statically built dependency graph. This is done in order to find main system constituents. Such an approach can be found in [212], [23], [310], [75], [143], [227], [172] or [331].

The second category consists of methods that are used to identify an a-priori known architectural pattern in the structure of the system. Such an approach is conceptually very similar to the methods discussed previously in this section. However, the definition of patterns refers to the substructures of the system that are relevant to its decomposition (e.g. use of Enterprise Java Beans) or concern very large, global patterns (e.g. decomposition into layers). Examples of such research can be found in [364], [196], [304] or [263].

The third category contains methods that enable an expert to understand the architectural structure of the analyzed system. This approach includes various formal query languages (e.g. [157], [181] or a very general view in [237]) statistics (e.g. [181]) and visualizations ([45], [46], [98] or [91]).

Architecture query languages

One of the aspects of an architecture mining task is the definition of convenient query language that can be used to find certain patterns in system structure. The approach described ranges from graph-defined queries, through various ontologies to predicate-based logical languages.

In [304], the author proposes an interesting approach for approximate mining of architectural patterns. The query in the Architecture Query Language (AQL) is given as a graph that represents the pattern to be found. The execution of the query is realized by finding the maximal assignment in the target graph with a queue of bounded length. According to the paper, the experimental results show that such a heuristic method is practically efficient, which means that the average accuracy of the pattern recovery evaluated on six mid-size software systems reached $65 - 70\%$. A similar approach, where software structure queries are given in the form of a graph, is given in [181].

In [193], the authors propose a universal language for describing changes in software repositories, which, basing on SCM logs, builds structural information on software entities modified in the commits. The general idea has also been discussed in [182], where the authors describe a set of different ontologies used to describe both structure and evolution of the software systems, or in [187] and [209], where *semantic web* technologies are used to combine and query a few data sources related to the software development process.

The model used in this thesis (see Section 6.1), though similar to the above, introduces a higher-order concept. A data fetching algorithm (described in Section 6.2), is based on the same input (SCM logs and issue tracker logs together), but the output produces a structure that represents the evolution of a dependency graph of the software system. We believe that this relatively small difference yields an important change in thinking about software evolution: It brings the analysis of software development process to the level of understanding changes in the complex object on both intra-entity and inter-entity modifications.

5.2.8 Defect prediction

Defect prediction is a classification problem in the domain of software systems. Its goal is to predict the number of *defects* (informally known as *bugs*)[3]

[3]We will use the term bug and defect interchangeably.

in particular areas of the software source code before they are identified and reported by users or testers. As reported in [195], typically as much as 10% of files are significantly more error-prone, than the remaining ones, and this observation has been confirmed by empirical studies on various software systems (e.g. [313]). This allows us to rephrase the goal of defect prediction in more approximate manner as the identification of a subset of software entities that contain significantly more defects than others. In the following sections we discuss a few diverse examples of defect prediction research, aimed on one of the two aforementioned goals.

A bug is typically fixed with some modification in the source code, which is introduced in a specific commit. Identification of commit which introduced the defect may be more challenging (see Section 5.2.8), but, simplifying it a bit, we may assume that there is usually a previous commit that actually introduced the bug into the code. This perspective allows us to consider bugs to be a spatio-temporal phenomenon in the software development process. Therefore, the methods described in this thesis can be potentially used as one of the methods of defect prediction. For more details, please refer to Section 6.7.4.

Complexity metrics

Research on defect prediction is often focused on the analysis of the correlation between various complexity metrics for a given entity and the number of defects identified or fixed in it. This is based on the intuitive assumption that more complex entities are harder to maintain and contain more defects. The research method typically falls under the same scheme: various metrics (both static and temporal) computed on the software entities are analyzed together with post-factum information about the number of defects present in it, most typically by analyzing bugs in the issue tracker. Then, statistical analysis is applied to such data in order to identify an association between metric values and error-proneness. Such an approach is described in [246], [245], [249], [309] or [274]. In-depth summary of these methods can be found in [175], [320] or [160].

Dependency graph analysis

A different approach is based on the analysis or specific traversal of dependency graphs, built from the system source code.

In [365], the author analyzes how changes done in one system are correlated to defects in systems that depend on it. The experimental data proves that there is such a correlation, and this phenomenon is called a *domino effect*. The research presented in the referred study is limited to the granularity of complete systems.

In [370] and [368], the authors propose a few methods of analyzing a fine-grained dependency graph for the prediction of software defects. The variety of approaches ranges from simple supervised learning on the adjacency matrix of the graph, to more sophisticated ones that use specific domain knowledge and dedicated metrics (called *network measures*) combined with statistical analysis. What is common to both methods is an experimentally proven hypothesis that estimating defect proneness of a given software entity can be done on the basis of the properties of its neighborhood. In fact, both papers argue that this is the approach that provides better prediction results.

In [166], the authors mine instances of anti-patterns in the source code and, by simple statistical analysis, show that the software entities that are either part of an anti-pattern or entities that depend on any part of an anti-pattern tend to have a higher number of defects.

A detailed survey of various approaches to defect localization based on graph mining techniques is also described in [107], [160] or [320].

In [195], the authors experimentally validate a hypothesis that bugs are introduced into the system "near" to each other. The nearness is defined in terms of 1) *changed-entity locality*, 2) *new-entity locality*, 3)*temporal locality* and 4) *spatial locality*, which correspond to specific respective hypotheses that new defects will appear in an entity that: 1) was recently changed, 2) was recently added, 3) had a defect recently fixed in it and 4) is dependent on another entity that had a defect recently fixed in it. The realization of the method is based on very simple concepts of *BugCache*, *FixCache*, *ChangeCache*, which are structures used to store recent bug fixes and modifications in the software source code. Despite its simplicity, the methods show to be very efficient in defect prediction. Depending on source data the authors report 73%-95% accuracy of predicting future faults at file level and 46%-72% at the level of the source code entity. Much simpler, yet similar approach is presented in [44]. In general, we can state that these two approaches are based on the concept of spatial and temporal nearness of bugs, which is similar to our framework presented in Section 6.5.

Defect origin analysis

Prediction of defects focuses on the estimation of number or, in more complex cases, distribution of defects in software, on the basis of some facts from the software development process. The *Defect origin analysis* is slightly different – it takes a more detailed look at the process and its main focus is to understand what was the reason for the introduction of bugs that have already been identified, and it is more related to the aforementioned *origin analysis*.

Elementary research in the topic, such as analysis of distribution in the software development process, shows that e.g. bugs tend to be introduced by larger changes (in terms of the number of lines modified in a single commit) (see [249]) or that more bugs are introduced on Friday (see [311]). Another, more advanced method is given in [41] or [194], where the authors use more semantic and low-level insight into the modifications made. Here the identification of the origin of a defect starts when the defect is actually fixed. The fix is identified by meta-data taken from the issue tracker and it is translated into a certain change in the source code viewed at as either text or abstract syntax tree. Once this is done, the algorithm scans the development history backwards to identify such a commit that actually made a reverse modification in the source code. Many variants of this method have been analyzed and experimentally validated (e.g. [298], [86] or [256]). The reported F-score spans from 0.44 to 0.77, but the method does not cope well with bugs which have external origin such as change in external API, or are actually introduced with multiple, unrelated changes to the program source code (see [299]). The bug origin analysis is used in the concept of spatio-temporal bug prediction mentioned in Section 6.7.4.

5.3 Resume

This chapter provides a selection of available literature on the topics related to this study. The first part describes canonical or generic data mining algorithms, which are directly or indirectly used in this thesis, and are listed at the beginning of this chapter.

The second part, starting from Section 5.2, discusses a narrower area of research in the general domain of mining software development process. It also provides a rough comparison of our study with related work, with explanation of some major differences and a few particular motivations and decisions. The most important ones are listed below:

- Acquisition of data from such software processes, where issue tracker SCM are technically synchronized (see Section 5.2.1).
- Limiting source code analysis to statical procedures only (see Section 5.2.1).
- A general, system- and programming-language-agnostic representation of source code structure that can be enriched with the additional information derived from its development process (see Section 5.2.1).
- A general concept of the adaptive algorithm and respective data structures used to analyze the software system development process (see Section 5.2.2).
- Identification of spatio-temporal patterns in the software development process that are correlated with general bad quality of the software system (see Section 5.2.1).
- A concept of the analysis of hierarchical, multi-level and approximate spatial patterns in the system source code (see Sections 5.2.3 and 5.2.7).
- Further automation of already described meta-concepts of software system analysis by replacing some tasks done by a human expert with appropriately fitted machine learning algorithm (see Sections 5.2.3, 5.2.4 and 5.2.6).

Chapter 6

Mining spatio-temporal rules in software evolution

This chapter describes the main contribution of this thesis: the proposed framework consists of a data representation model with appropriately suited mining algorithms, which are used to detect spatio-temporal patterns in the software development process. A reader should be familiar with the concepts explained in Chapters 2, 4 and 5. In the following sections you will find a detailed description of:

- the algorithms used in this research to acquire the data from software repositories and other systems used in the software development process.
- the abstract model for representing the structure and evolution of a software system,
- the adaptive algorithm for building and updating the model along with the process,
- the machine-learning-based methods for detecting spatio-temporal patterns in the software development process,
- proposals of applications of the framework to other domains,
- a reference to the experiments that were carried out to validate efficiency of the proposed methods.

6.1 Data representation

This section describes in detail the proposed model for the representation of the software development process and discusses its crucial properties.

6.1.1 Software snapshot

Software snapshot is a formal model that represents a static version of the software system source code at a given point in time. If we treat the developed software system as a complex object, the software snapshot may be viewed as a state of a complex object. A time-labeled sequence of software snapshots models the software evolution process.

Given the source code of a system at a given revision r, let Ent_r be the set of all code entities present in it. Technically, since this research is based on the systems implemented in Java, and the granularity of the source code analysis ranges from packages (the most coarse-grained) up to methods and fields (the most fine-grained), we put: $Ent_r = Pack_r \cup Files_r \cup Class_r \cup Meth_r \cup Field_r$, where: $Pack_r$ denotes the set of all packages, $Files_r$ denotes the set of all source files, $Class_r$ denotes the set of all classes (including interfaces), $Meth_r$ denotes the set of all methods and $Field_r$ denotes the set of all fields present in the software source code at revision r.

Let $Dep\_Types = \{contain, parameter, extend, call, implement, refer, type, variable\}$ denote the types of dependencies between software entities. $Deps_r \subseteq Ent_r \times Ent_r \times Dep\_Types$ is a multi-relation[1] that represents the edges of a labeled multigraph of dependencies between software entities from Ent_r. For better comprehension we will define $Deps_r$ as a relation, while keeping in mind that each tuple in this relation (which corresponds to a single relation between two code entities described in Definition 19) has its arity which can be greater than one. Consequently, the same two entities can be in the same type of relation multiple times (e.g. one method can call another method more than once, one method can have two different parameters of the same type, etc.).

Definition 19. Dependency relation $Deps_r \subseteq Ent_r \times Ent_r \times Deps\_Types$ *is given by:*

[1]Multi-relation is a multi-set of tuples (as opposed to set of tuples in canonical relation). Consequently it encodes a multigraph instead of graph.

- *A tuple $(e_1, e_2, contain) \in Deps_r$ iff the source code of the entity represented by e_2 is contained in the source code of the entity represented by e_1 or when a type (class or interface) represented by e_2 is contained in package represented by e_1,*

- *a tuple $(e, c, variable) \in Deps_r$ iff the body of the entity represented by e declares a variable of type represented by class c,*

- *a tuple $(m, c, parameter) \in Deps_r$ iff the method represented by entity m declares a formal parameter of type represented by c,*

- *a tuple $(c_1, c_2, extend) \in Deps_r$ iff the class represented by c_1 is a subclass of the class represented by c_2,*

- *a tuple $(c_1, c_2, implement) \in Deps_r$ iff the class represented by c_1 is an implementation of the interface represented by c_2,*

- *a tuple $(e, m, call) \in Deps_r$ iff the body of the entity represented by e contains a call of a method represented by the method entity m,*

- *a tuple $(e, f, refer) \in Deps_r$ iff the body of the entity represented by e contains a reference to the field represented by the field entity f,*

- *a tuple $(f, c, type) \in Deps_r$ iff c represents a class that is a declared type of the field represented by f or a declared return type of a method declared by f.*

Let $\mathscr{L}_{Java}$ denote the set of strings that represent valid pieces of Java source code that define a source code entity, and let R denote all revisions of an analyzed system. The function $Source_r : \bigcup_{r \in R} Files_r \cup Class_r \cup Meth_r \rightarrow \mathscr{L}_{Java} \cup \{\perp\}$ encodes the source code of entities according to the definition 6.1:

$$Source_r(e) = \begin{cases} \perp & \text{if } e \text{ is not present at revision } r, \\ \text{the source code of } e \text{ at revision r} & \text{otherwise.} \end{cases} \tag{6.1}$$

For a fixed finite set of software metrics M, $Metr_r : (Files_r \cup Class_r \cup Meth_r) \times M \rightarrow \Re \cup \{\perp\}$ is a function that encodes the values of the metrics for applicable source code entities available at revision r and is given by:

$$Metr_r(e, m) = \begin{cases} \perp & \text{if } Source_r(e) = \perp \\ m(Source_r(e)) & \text{if } m \text{ is applicable to } e, \\ \perp & \text{otherwise.} \end{cases} \tag{6.2}$$

The $\perp$ used in the above definitions is a special symbol that conceptually corresponds to non-existing context (e.g. a source code of non-existing file does not exist or a metric value for non-existing source code does not exist).

Basing on the preceding definitions we say that a *software snapshot* at revision r is a tuple :

$$SSn_r = (Ent_r, Deps_r, Metr_r)$$

Please note that SSn_r can be treated as a vertex-labeled and edge-labeled multigraph (*property graph*) where:

- Ent_r is a set of vertices, and each vertex is uniquely identified by the absolute name and type of an entity it corresponds to (e.g. 'java.lang.String', class)) ,
- $Deps_r$ is a multi-set of labeled edges, with labels from Dep_Types. We will say that entities e_1 and e_2 are connected by an edge with label t iff $(e_1, e_2, t) \in Deps_r$.
- $Metr_r$ is the vertex-labeling of the graph, where the label of the node is a vector of the values of its metrics. If $M = \{m_1, \ldots, m_n\}$, then the label of entity e is the vector $(Metr_r(e, m_1), \ldots Metr_r(e, m_n))$.

A simplified structure $(Ent_r, Deps_r)$ without metrics is an edge-labeled multigraph.

For the sake of simplicity the $Deps_r$ relation will be decomposed in the following binary relations according to the following definitions:

- $(e_1, e_2) \in Cont_r$ iff $(e_1, e_2, contain) \in Deps_r$,
- $(e_1, e_2) \in Param_r$ iff $(e_1, e_2, parameter) \in Deps_r$,
- $(e_1, e_2) \in Ext_r$ iff $(e_1, e_2, extend) \in Deps_r$,
- $(e_1, e_2) \in Call_r$ iff $(e_1, e_2, call) \in Deps_r$,

- $(e_1, e_2) \in Imp_r$ iff $(e_1, e_2, implement) \in Deps_r$.

- $(e_1, e_2) \in Ref_r$ iff $(e_1, e_2, refer) \in Deps_r$,

- $(e_1, e_2) \in Typ_r$ iff $(e_1, e_2, type) \in Deps_r$,

so that $Deps_r = Cont_r \times \{contain\} \cup Param_r \times \{param\} \cup Ext_r \times \{extend\} \cup Call_r \times \{call\} \cup Imp_r \{implement\} \cup Ref_r \times \{refer\} \cup Typ_r \times \{type\}$.

The $\mathcal{SSn}$ will denote the set of all possible triplets of such a construction, which correspond to the formally valid source code of a program written in Java. In the above definitions, the SSn structure, indexed by revision r, reflects the structure of the complete source code of an analyzed system at revision r. Other elements of $\mathcal{SSn}$ are used further in this chapter and the same notation is used in these cases. For any $SSn_? \in \mathcal{SSn}$, the sub-sets $Pack_?, Files_?, Class_?, Meth_?$ and relations
$Deps_?, Cont_?, Param_?, Ext_?, Impl_?, Call_?, Ref_?, Typ_?$ will be defined likewise. The mutual correspondence should be clear from context and by convention all symbols will share common upper and lower indexes. For example $Meth_i^{add}$ will denote the set of entities representing the methods from the multigraph SSn_i^{add}

6.1.2 Software evolution

Please recall from Section 2.2 that it is possible to extract such a fragment of a software development process, which represents consistent work on the system arranged along a linear timeline. Technically it is represented by a linearly ordered set of revisions R. In such a setting, the series $(SSn_r)_{r \in R}$ is a formal model for *software evolution*.

Conceptually, a *software snapshot* is a structure that contains information about all packages, classes and methods, their properties and inter-relations relevant for this research at a given revision. Thus, *software evolution* is a representation of changes in this structure that take place during the software development process. Such a model is used as the basis of the theory and the experiments discussed in this thesis. In the following paragraphs you will find more technical information on how it is built from the SCM logs.

6.2 Raw data fetching

The following subsections explain technical means used to construct software evolution from the source code.

6.2.1 Single revision parsing

This subsection describes the process of constructing SSn structure for a given, fixed revision r.

Software snapshot evaluation context

Usually the source code of any program can rely on additional libraries, which are provided in a pre-compiled form, so their source code is not always available. Yet, the structure of software in such libraries must be known to such an extent that allows it to be referred to from the source code of other systems. In particular, in Java, all packages, classes, methods and fields, as well as their mutual relations, are known. It means that relations of type $\{contain, parameter, extend, type\}$, as defined in Section 6.1.1, between software entities originating from the external library are known. Therefore, the structure of external libraries, whose entities may be referred to from the source code of the analyzed system, can be represented by a structure similar to the SSn, with the contents limited only to the information that is actually available. We will call such structure the *context* and denote it by Ctx. Formally

$$Ctx = (Ent_c, Deps_c, Metr_c),$$

where $Ent_c, Deps_c$ and $Metr_c$ are defined in the same way as for the *software snapshot* in Section 6.1.1, but:

- if a metric m for entity e is unknown, then $Metr_c(m, e) = \{\perp\}$
- if relation $d \in Dep\_Types$ is unknown then $Deps_c \cap (Ent_c \times Ent_c \times \{d\}) = \emptyset$

The following paragraphs describe the method of constructing SSn_r from the source files of the system at revision r, in a given context Ctx.

Source files

First, all sub-folders that contain a non-test source code written in Java are scanned for files with extension `.java`. A list of such files directly yields the set $Files_r$. Technically, the set of folders with non-test files is in-general evaluated by a heuristic that takes only such file whose path satisfies the following conditions: 1. It contains sub-strings `''/src/''` and `''/java/''` 2) 2. does not contain sub-string `''/test/''` before sub-string `''/java/''`. On rare occasions the set of source folders can be adjusted to the particular source code layout. In such case it becomes part of the configuration for a given data set. The problem of identifying the source files in a given software system is a trivial technical task and will not be considered further in this thesis.

All files from $Files_r$ are parsed and the contents of each is represented in the form of a abstract syntax tree for further scanning. The function $Source_r$ defined in equation 6.1 is used to retrieve the source code of a file and the abstract syntax tree is computed from the source code according to the language specification ([260]).

Packages

If the abstract syntax tree derived from file f contains a node with a package declaration, the corresponding element p is added to the $Pack_r$ set and the $Deps_r$ relation is extended with respective *(p, f,* contain*)* tuple.

Classes and Fields

Further on, the abstract syntax tree is scanned for nodes that represent a declaration of a Java class or interface (including inline and anonymous ones). This is done so that appropriate elements can be added to $Class_r$ and $Deps_r$ can be extended with appropriate *(*c_1*,* c_2*,* contain*)* triplets. Additionally, if the class declaration contains a reference to a super-class or interface, appropriate (c_1, c_2, *extend*) or (c_1, c_2, *implement*) tuple is added to the $Deps_r$ to represent this fact. The c_2 may not be defined in any source code file and may come from Ctx (i.e. $c_2 \in Ent_c$). In such case c_2 is added to the Ent_r.

If the class contains a declaration of a field, then a new element is added to $Field_r$ and an appropriate tuple (c, f, *contain*) is added to $Deps_r$, where c represents the currently parsed class and f represent the field. Since the type of the field must be provided in the declaration, then appropriate tuple $(f, t, type)$ is added to $Deps_r$, where f represents the field and t represents the type of the field but only when t is represented in the source code of the analyzed system or it is represented by an entity from Ctx, i.e. $t \in Ent_c$. In the latter case, t is added to the Ent_r.

Methods

In the course of further abstract syntax tree traversal, for each class c, nodes representing method declaration within it are translated into elements of $Meth_r$ and appropriate tuples (c, m, *contain*) are added to $Deps_r$, where c and m represent, respectively, the class and the method in this class. Technically, the proposed model contains a simplification: Within a single class, all overloaded methods with the same name and the same number of arguments (called *equinominal methods*) are contracted to one element of $Meth_r$. This is motivated by the fact that, typically, if a class contains two overloaded methods, they functionally represent the same operation but, due to technical reasons, depend on different arguments (see [38]). With additional discrimination based on the number of parameters, this simplification produces very insignificant error (see Appendix A.1.1).

Lastly, the header of each method m is analyzed so that:

1. if the method returns a type represented by the class entity c, the tuple (m, c, *type*) is added to $Deps_r$, if c is either represented in a source code file or it comes from the context,

2. if the method contains a formal parameter of a type represented by the class entity c and c is represented in the source code of the analyzed system or comes from the context, the tuple (m, c, *parameter*) is added to $Deps_r$,

In both cases the c may not be defined in any source code file but may come from Ctx. In such a situation, c is added to Ent_r. The procedure of finding method entities has some practical simplifications: 1. methods that override methods from java.lang.Object, such as toString, hashCode and equals are ignored; 2. constructors are treated as methods, with no return type (which is other than java “void” type).

Call graph and field references

The last part of the abstract syntax tree traversal is scanning all the body blocks inside methods and classes. This translates each AST node that represents a method call or a field reference into respective tuples in $Deps_r$ relation according to the following rule: If the body of the inner-most entity e, for each maximal AST sub-tree that corresponds to the java expression that can be written as $\varepsilon_1.\varepsilon_2.(\ldots).\varepsilon_n$, where each ε_i corresponds to either a class, variable, field or a method call, then such expression is analyzed to extract appropriate $Deps_r$ tuples. Please note that in valid Java program source code: 1. such a decomposition is unambiguous, 2. the declared type of each variable/field is known and we know this type of inheritance hierarchy, 3. the declared return type of each method is known (we will call it the type of method for the consistency of notation) and we know this type of inheritance hierarchy. For each sub-expression $\varepsilon_{i-1}.\varepsilon_i$, given that the type of ε_{i-1} is represented by a software entity c_{i-1}:

- if the ε_i corresponds to a field in c_{i-1}, represented by a software entity f, a pair (e, f) is added to Ref_r.
- If ε_i corresponds to a method of c_{i-1}, represented by a software entity m, then a pair (e, m) is added to $Call_r$.

Similarly:

- If ε_1 is a field represented by an entity f then (e, f) is added to Ref_r.
- If ε_1 is a method represented by an entity m then (e, m) is added to $Call_r$.

When the type of any ε_i corresponds to a type that is not represented in the Ent_r (i.e. it is neither defined in the source code nor in the context), then the analysis of such an expression is skipped and, consequently, the information about inter-entity dependencies coming from it is lost.

The above procedure is based on the following simplifying assumptions:

- **Call graph analysis based on declared types only**.
  The called method is inferred only by evaluating the *declared type* of the called object. Given the example from Listing 6.1:

Listing 6.1: Declared vs. runtime type

```
1 public class MyClass {
2   public void myMethod() {
3     Number num = new BigDecimal();
4     num.toString(); // <- actual call
```

the method call in line 4 is translated to the entry that represents the fact *"method myMethod of class MyClass calls a method toString of a class Number"*, because the *declared type* of variable num is Number, even though it is clear that the actual type is BigDecimal - a subclass of Number. The motivation for such a simplification comes from the fact that the actual type of the callee cannot always be determined at compilation time. Therefore, a consistent approach is taken to consider the declared type, since this information is always available during the static code analysis.

- **Contraction of equinominal methods**.
  As mentioned above, the equinominal methods are contracted to a single software entity. This also influences the call graph reconstruction. For example, the following code excerpt:

Listing 6.2: Equinominal methods

```
public class MyClass {
  public void myMethod() {
    BigDecimal.valueOf("123");
    BigDecimal.valueOf(123L);
```

is translated into such a subgraph that represent the fact *"method myMethod of class MyClass calls a single method valueOf of class BigDecimal twice"*. The justification for this simplification is given above, but also comes from experimental validation: It turns out that the fraction overloaded methods among all methods can exceed 4%, but fraction of overloaded methods with the same number of arguments does not exceed 0.75% (see Table A.1). A problem can arise if two equinominal methods declared in a single class return two different types. In such case it is impossible to determine a corresponding pair in Typ_r relation. In order to cope with such an ambiguity, the following rule is used: The return type of a method entity, which corresponds to more than one actual method declaration in the source code, is set to the most specific class, that is, a super-class of all return types used in such a declaration. For example, the code from excerpt 6.3 produces a single method entity named *myMethod* with return type `java.lang.Number` as it is a common super-class of `Double` and `Integer`.

Listing 6.3: Equinominal methods with different return type

```java
public class MyClass {
  public Double myMethod(String s){...}
  public Integer myMethod(Number s){...}
```

In practice the problem of equinominal methods with different return type is negligible, as such situations happen very infrequently. The experiment described in A.1 shows that the number of equinominal methods with different return types usually does not exceed 1‰ of the total number of methods (see experimental validation in Appendix A.1.1). Similar simplification is also proposed in [60].

- **Pruned references to external entities**.
The source code of a program written in Java may depend on classes that are defined outside of it, usually in some external library. The listing 6.2 is a good example here: the class java.math.BigDecimal used in the code comes from the Java runtime library and is never defined in the source code. The class or method that is referenced in the program source code but is not defined in it is called respectively: *external class* and *external method*, most generally: *external entity*. Please note that external method can only be contained in an external class. As the research described in this thesis relates to the identification of spatio-temporal patterns only within the developed system, the references to external libraries are irrelevant. Therefore, once the SSn_r graph is constructed, all nodes that originate from external libraries are removed, together with all edges adjacent to them. We will call this procedure *pruning references to external entities*. It allows us to simplify the model: Thus, in the following considerations we omit external entities entirely. We assume that entities in any software snapshot and context come only from the source code of the analyzed system. For technical reasons in two specific cases we preserve information that could be lost after external references are pruned. Namely: when a method return type is void or if a method is a constructor. This is done so that we can

discern these cases from a situation when a method returns an external type that was pruned.

Software Metrics evaluation

The function $Metr_r$ encodes the values of certain software metrics for all applicable entities from Ent_r. Some of them have been mentioned already in Section 2.2.7. The following list contains all metrics used in this research and briefly describes their definition and the technical means of their calculation. If not stated otherwise, the computation of the metrics was done with the use of *Checkstyle* - a tool for static code analysis. For details, please see [6].

- **Data abstraction coupling**
  This metric is applicable for class entities and measures how many instances of other classes are instantiated within the source code of a given class. The higher the value is, the more direct external dependencies the class has, making it more unstable.

- **Fan out**
  A metric similar to *Data abstraction coupling*, which measures the number of classes a given class depends on. Again, high number of such dependencies makes the class more vulnerable to modifications in other areas of the source code, which makes it more unstable.

- **Cyclomatic complexity** and **NPath complexity**
  These two metrics measure the complexity of a code block. Cyclomatic complexity, based on the classic work [230], denotes the number of decision point instructions within the body block increased by 1. The NPath complexity (see [254]) denotes the theoretical maximum number of different acyclic execution paths that could go through the code block. Both metrics measure how complex the structure of the source code fragment is. Clearly, the difficulty of understanding and maintaining the code increases with their value.

- **Number of lines of source code in entity**
  This simple metric measures how many lines of code a given file, class or method take. Technically, the evaluations do not count empty lines or lines with comments, so that it approximates the actual size of the respective source code fragment.

- **Lack of cohesion of methods (LCOM1, LCOM2, LCOM3, LCOM4, TCC)**
  LCOM is a suite of software metrics that evaluate the design of a given class by quantitative analysis of relations between its methods and fields (see [78]).

  LCOM1 is defined as the difference between powers of two sets: the set of all pairs of different methods that use a non-empty disjoint set of class fields and the set of all pairs of different methods that use at least one field altogether. If the result is negative, the value of this metric is set to 0.

  LCOM2 and LCOM3 metrics, defined for the class and the formulae to evaluate them, are based on the following notions: m - the number of methods in a class, A - set of fields of a class, m_a the number of methods that refer to field a:

  $$LCOM2 = 1 - \frac{\Sigma_{a \in A} m_a}{m * |A|}$$

  $$LCOM3 = \frac{m - \frac{1}{|A|}\Sigma_{a \in A} m_a}{m - 1}$$

  LCOM2 corresponds to the fraction of methods that do not refer to a specific field normalized over all fields. LCOM3 is similarly normalized with respect to fields and methods.

  LCOM4 (see [158]) is expressed in terms of a graph of inter-method dependencies. We say that two methods are adjacent iff one of them calls the other one or there is at least one field used by both methods. The number of connected components in such a graph is the value of LCOM4.

  Conceptually, all these metrics measure to what extent the behavior of the class (methods) depends on its state (fields). If all methods depend on all fields, then the value of all metrics reaches its minimum. The values grow when a class can be easily divided to independent parts, operating on a separated subset of fields and providing a separated subset of methods, each having own, separated responsibility. Such a situation may indicate a design smell - a violation of *principle of single responsibility* for a class, or simply *lack of cohesion in methods.*

Instead of looking at relations between methods and fields within a single class, we can also measure the relation between its methods in a similar manner, with the use of Tight Class Cohesion (TCC) metric (see [52]). It is defined as the number of pairs of methods that invoke each other divided by the number of all such pairs.

Technically, the value of LCOM and TCC metrics can be evaluated during the analysis of the abstract syntax tree of a single class. Clearly, only two types of information must be extracted: the inter-method call and the use of a variable within the body of the method. The former is part of $Call_r$ set and the latter is available in Ref_r. If one of the above formulae is undefined (e.g. when a class does not have fields), the respective metric value is $\perp$ (see Section 6.1.1).

All the above metrics can be directly evaluated solely from the source code of the entity they apply to. They therefore satisfy the property described in section 2.2.7, where each metric is defined as the function that, given the source code of an entity, produces the value of the metric for this entity. This fact is significant for the method of adaptive evaluation of function $Metr_?$ described in section 6.2.3.

Metrics not derivable from the source code There are some popular software metrics that cannot be evaluated solely from the source code of the respective entity, as they require some additional context information. Two such examples are:

- **Depth of inheritance tree (DIT)**
  This metric is applicable to the class only and it measures the number of nodes on the path in the inheritance tree from the node representing the `java.lang.Object` class to the node representing the given class. Large value of this metric indicates a deep inheritance tree, which might indicate the presence of *Yo-yo* design anti-pattern . Technically, the value of this metric can be calculated directly from the Ext_r set.

- **Fan in (FI)**
  A metric dual to the *Fan out*. It measures the number of other classes that depend on a given class. The greater the value, the more likely it is that a change in the class will affect other fragments of the software source code.

According to what has been explained in the previous paragraph, such metrics are not encoded by $Metr_?$ function. Yet, in practice their value can be easily evaluated in the proposed model. Indeed: For a given class c represented by a node $n \in Ent_?$:

- DIT is the number of nodes on the longest path starting at n and built only from the edges of type *extend*. This path is usually very short and its length typically does not exceed 7 (see Appendix A.4.3).
- FI is the number of other nodes of the generalized call graph (see Definition 4), which are connected with a direct edge to n. It is sufficient to traverse only direct neighbors of n to compute this metric.

Since the value of the two metrics can be trivially and efficiently computed once the SSn graph is constructed, we will typically assume, for the sake of consistency, that $Metr$ can encode all metrics. On occasion, when this distinction makes an important difference, DIT and FI will be treated separately.

6.2.2 Single revision parsing formalism

For a given fixed SCM and the aforementioned experiment configuration with the list of source folders, the procedure of building the SSn_r can be formally represented by a function that, given the revision r, returns the tuple SSn_r. In fact, this function is actually expressed as a deterministic program that is part of this thesis and is described in Appendix B.5.

Please recall that the procedure starts with determining the set of files that yield the $Files_r$ set, which is the only input for later phases of the process of building the SSn_r structure. It means that the aforementioned program can be split into two subsequent subprograms: the first, which finds the set of files, and the second one, which builds upon it a SSn_r by analyzing the contents of the files.

Therefore, if we denote: by $\mathscr{F}$ the set of all possible source files in the evolution of analyzed system and by $\mathscr{L}_{Java}$ the set of strings that correspond to a valid source code of code entity in Java[2], we may assume that there is a function $FP : P(\mathscr{F}) \times {\mathscr{L}_{Java}}^{\mathscr{F}} \times \mathcal{SSn} \rightarrow \mathcal{SSn}$ that, given the set of files F and a context, builds a tuple

$$FP(F, Source, Ctx) = (F \cup Ent_F, Deps_F, Metr_F) \tag{6.3}$$

[2]same as in section 6.1.1

according to the algorithm described in section 6.2.1. Here Ent_F is a set of entities declared in source code files F. The function FP represents the procedure of parsing, interpreting and transforming the list of source code files into corresponding SSn structure. Clearly, for any revision r

$$FP(Files_r, Source_r, (\emptyset, \emptyset, \emptyset)) = SSn_r.$$

The function FP is used for building the evolution model described in the following sections.

6.2.3 Fetching evolution data

The previous section explains technical means to build a *software snapshot* for a given fixed revision. Given that we have a method for finding a linearly-ordered set of revisions R, the naive method for computing the software evolution is straightforward and entails evaluating SSn_r for each $r \in R$ separately. However, the algorithm can be more efficient, as it can make use of the fact that R is linearly ordered and usually the changes between two subsequent revisions are limited (see experimental results in A.2). The following subsections describe the means used to compute R and $(SSn_r)_{r \in R}$ in such a manner.

Extracting revisions list

The method of determining the list of revisions used in this research is trivial, based on the fact that all the projects that served as a source of data for the experiments used the concept of the main development branch (see [106]). Accordingly, the SCM of the project contains one dedicated branch that represents the main line of system development history (*main branch).* All changes that are being developed in other branches must be eventually merged to the main branch, so that all commits done in all other branches become visible in the main branch. Therefore, if we omit all other branches, we can consider that SCM contains only one main branch, with perfectly linear order on the commits. In such a setting the method for determining R is straightforward: Given a SCM and the main branch, we need to take from it all the commits. We assume that the source code at each revision is formally correct. Therefore, if the source code does not compile at some revision due to some bug, we treat such a revision as non-existent.

Adaptive software snapshot evaluation

A single commit usually modifies a tiny fragment of the system source code. This fact has been experimentally validated in experiments described in Appendix A.2, which show that in the analyzed software history $76 - 99\%$ (depending on the software project) commits do not modify more than 10 files. Moreover, the average number of modified files ranges from 3.97 to 16.47, depending on the project.

Consequently, we might expect with great probability that two subsequent software snapshots are to a large extent identical. Therefore, one can be derived from the other by computing only the tiny difference between them. This concept is formalized below.

Evolution parsing formalism In the preceding paragraphs we considered a formalism that describes the state of a software source code at a given point in time. Technically, the changes done in subsequent commits modify the files that constitute the source code, and the location and contents of these files define the actual logical structure of the source code that yield the complete SSn. To be able to track the changes, we must discern the unique identity of each code entity from its internal contents. For example, when a file content is modified within a commit in such a way that the name of the class defined in it is modified, then logically, the class with the old name is removed, the class with a new name is added, while the file is present in both versions. The following list contains the formal definitions of identity of each code entity present in a software snapshot.

1. Packages are uniquely identified by their full name.

2. Files are uniquely identified by their path, relative to the path of the source folder they belong to.

3. Classes, including anonymous[3], nested and inner classes, are uniquely identified by their full name, consisting of the full name of the software entity they are directly contained in and their local name.

4. Methods and fields are uniquely identified by their full name consisting of the full name of the entity they are declared in, their local name and the number of their formal parameters (in case of methods).

[3]In Java anonymous classes have a unique technical name ([260]).

Listing 6.4: Example of unique identity of method

```
package p;
class MyClass {
  class MyInnerClass {
   public void myMethod() {}
   public void myMethod(int i) {}
  }
}
```

Example According to the above rules the two methods declared in Listing 6.4 are identified by 'p.MyClass#MyInnerClass#MyMethod#0' and 'p.MyClass#MyInnerClass#MyMethod#1' respectively.

Files affected by a commit Assume that r_1 and r_2 are two subsequent revisions from the software development process and SSn_{r_1} and SSn_{r_2} are the respective software snapshots. Let $Files^{add}_{r_1,r_2} = Files_{r_2} \setminus Files_{r_1}$, denote the set of software source code files that were added in r_2. Let $Files^{rem}_{r_1,r_2} = Files_{r_2} \setminus Files_{r_1}$, denote the set of software source code files that were removed at revision r_2. Let $Files^{mod}_{r_1,r_2} \subseteq Files_{r_1} \cap Files_{r_2}$ denote the set of source code files whose contents were modified in r_2.

A transitive closure of a binary relation R will be denoted R^*. The notation $Deps^*_? \subseteq Ent_? \times Ent_? \times Dep\_Types$ will be called the transitive closure of $Deps_?$ relation and will denote: $Deps^*_? \overset{def}{\equiv} Cont^*_? \times \{contain\} \cup Param^*_? \times \{parameter\} \cup Ext^*_? \times \{extend\} \cup Imp^*_? \times \{implement\} \cup Call^*_? \times \{call\} \cup Typ^*_? \times \{type\} \cup Ref^*_? \times \{refer\}$.

Conceptually, a file *affects* another if changes done in the first one may change the semantics of the source code of the second one. We will limit the semantics only to the construction of the software snapshot, as defined in Section 6.2.2.

Definition 20. *We will say that file f_1 is* affected *by file f_2 at revision r iff $f_1 = f_2$ or there are software entities e_1, e_2 such that $(e_1, f_1) \in Cont^*_r$ and one of the following conditions is met:*

1. $(e_2, f_2) \in Cont^*_r$ *and* $(e_1, e_2) \in Call_r \cup Refs_r$

2. there are classes c_1, c_2 such that (e_2, f_2), $(e_2, c_1) \in Cont^*_r$ and $(c_1, c_2) \in Ext^*_r$.

For a given set of files $F \subseteq Files_r$ the set of files affected by F at revision r will be denoted $Aff_r(F) = \bigcup_{f \in F}\{x : x \text{ is affected by } f \text{ at revision } r\}$.

Definition 21 (Affected entity)**.** *Let $Files^{mod}_{r_1,r_2}$ and $Files^{rem}_{r_1,r_2}$ denote the set of files whose contents was modified between revisions r_1 and r_2 and the set of files which were removed between revision r_1 and r_2 respectively. Let $Files^{aff}_{r_1,r_2} = Aff_{r_1}(Files^{mod}_{r_1,r_2} \cup Files^{rem}_{r_1,r_2})$ denote the set of files affected at revision r_1 by files that were modified or removed in r_2.*

We will say that entity e is affected *by revision r_2, which follows revision r_1 if it is transitively contained in file f (i.e. $(e, f) \in Cont^*_r$) and $f \in Files^{aff}_{r_1,r_2}$.*

Adaptive evaluation of software snapshot The following paragraphs lead to the conclusion that SSn_{r_2} can be adaptively and efficiently evaluated on the basis of preceding SSn_{r_1} with the use of information about files that were added or removed or whose contents changed at revision r_2.

We will introduce a special software snapshot $SSn^{\Delta}_{r_1,r_2}$, which will conceptually represent a fragment of the structure of the source code that changes between revisions r_1 and r_2.

Definition 22. *A (r_1, r_2)-$differential$ snapshot is defined as:*

$$SSn^{\Delta}_{r_1,r_2} = FP(Files^{aff}_{r_1,r_2} \cup Files^{add}_{r_1,r_2} \setminus Files^{rem}_{r_1,r_2}, Source_{r_2}, SSn_{r_1})$$

In this definition, the function FP defined in Section 6.2.2 is used with $Source_{r_2}$ as a second argument and SSn_{r_1} as a context argument. It means that we evaluate a fragment of the source code (precisely: a source code of affected and added files) at revision r_2 in the context of a software snapshot from revision r_1. Since the context is represented by SSn_{r_1}, no entities from it come from the external library. Therefore, these entities are not pruned from $SSn^{\Delta}_{r_1,r_2}$, as it is described in Section 6.2.1.

Theorem 1. *SSn_{r_2} can be constructed from $SSn_{r_1}, Files^{add}_{r_1,r_2}, Files^{rem}_{r_1,r_2}, SSn^{\Delta}_{r_1,r_2}$ without use of $Source_{r_2}$ function.*

Proof.

Claim 1.1. *Ent_{r_2} can be determined from $SSn_{r_1}, Files^{add}_{r_1,r_2}, Files^{rem}_{r_1,r_2}, SSn^{\Delta}_{r_1,r_2}$*

Proof of Claim 1.1.

Clearly, $Files_{r_2} = Files_{r_1} \cup Files^{add}_{r_1,r_2} \setminus Files^{rem}_{r_1,r_2}$.

1) Entity $e \in Pack_{r_2}$ *iff there is a file* $f \in Files_{r_2}$ *such that* $(e, f) \in Cont^*_{r_2}$. *It is possible if one of the following conditions is met:*

1. $f \notin Files^{\Delta}_{r_1,r_2} \wedge f \in Files_{r_1} \wedge (e, f) \in Cont^*_{r_1}$
2. $f \in Files^{\Delta}_{r_1,r_2} \wedge (e, f) \in (Cont^{\Delta}_{r_1,r_2})^*$

Since all the above conditions are directly encoded in SSn_{r_1} *or* $SS^{\Delta}_{r_1,r_2}$ *and they are mutually exclusive (since a file either is or is not a member of* $Files^{\Delta}_{r_1,r_2}$*),* $Pack_{r_2}$ *is determinable from these structures.*

2) Entity $e \in (Class_{r_2} \cup Meth_{r_2} \cup Field_{r_2})$ *iff there is a file* $f \in Files_{r_2}$ *such that* $(f, e) \in Cont^*_{r_2}$. *It is possible if one of the following conditions is met:*

1. $f \in Files^{\Delta}_{r_1,r_2} \wedge (f, e) \in (Cont^{\Delta}_{r_1,r_2})^*$
2. $f \notin Files^{\Delta}_{r_1,r_2} \wedge f \in Files_{r_1} \wedge (f, e) \in Cont^*_{r_1}$

Similarly, all the above conditions are directly encoded SSn_{r_1} *or* $SS^{\Delta}_{r_1,r_2}$ *and they are mutually exclusive. Therefore* $Class_{r_2}$, $Meth_{r_2}$ *and* $Field_{r_2}$ *are determinable from these structures.*

Since $Pack_{r_2}, Files_{r_2}, Class_{r_2}, Meth_{r_2}$ *and* $Field_{r_2}$ *can be determined from* SSn_{r_1} *and* $SSn^{\Delta}_{r_1,r_2}$ *, so can the entire* Ent_{r_2}. ■

Claim 1.2. $Deps_{r_2} \setminus (Ent_{r_2} \times Ent_{r_2} \times \{calls, refer\})$ *can be determined from* $SSn_{r_1}, Files^{add}_{r_1,r_2}, Files^{rem}_{r_1,r_2}, SSn^{\Delta}_{r_1,r_2}$

Proof of Claim 1.2. *The relation* $Deps_?$ *is defined in such a way that for any entities* e_1, e_2 *and* $t \in Dep\_Types \setminus \{call, refer\}$, *such that* e_1 *is not a package, the* $(e_1, e_2, t) \in Deps_?$ *can be determined only by the analysis of the source code of the file in which* e_1 *is contained, given that* e_1 *and* e_2 *are known up front. Indeed: In Java, if a class* c_1 *extends another class* c_2, *then this relation must be declared within the source code of* c_1, *(in the file in which* c_1 *is declared) and this declaration must unambiguously point at* c_2. *The same argument also works for class implementing an interface and the interface extending an interface. Similarly: a declaration of field or variable, method return type or parameter formal type, must contain an unambiguous type* *of the respective code entity. From the above facts we can conclude that if a software entity e_1 is not a package and there is an entity e_2 such that $(e_1, e_2) \in Cont^*_{r_2}$, then e_1 and e_2 must be declared in the same source file. Therefore, if an entity e_1 is not a package and it is transitively contained in a file $f \in Files_{r_2}$, then all relations (e_1, e_2, t), such that $t \in Dep\_Types \setminus \{call, refer\}$ are defined in $Deps_{r_1}$, if $f \notin Files^{\Delta}_{r_1,r_2}$ or in $Deps^{\Delta}_{r_1,r_2}$ otherwise. It means that $Deps_{r_2} \setminus ((Ent_{r_2} \setminus Pack_{r_2}) \times Ent_{r_2} \times \{call, refer\})$ can be determined from $SSn_{r_1}, Files^{add}_{r_1,r_2}, Files^{rem}_{r_1,r_2}, SSn^{\Delta}_{r_1,r_2}$*

According to the definition from Section 6.2.2 each source code file can contain at most one definition of a package, which uniquely determines the contain *relation for the file. Moreover, if $(p, f, t) \in Deps_{r_2}$ and p is a package, then: 1. f is a file, 2. $t = contain$. Therefore for any package p, $p \in Pack_{r_2}$ if one of the following conditions is met:*

1. *There is a file $f \in Files_{r_1} \setminus (Files^{rem}_{r_1,r_2} \cup Files^{\Delta}_{r_1,r_2})$ and $(p, f, contain) \in Deps_{r_1}$*

2. *There is a file $f \in Files^{\Delta}_{r_1,r_2}$ and $(p, f, contain) \in Deps^{\Delta}_{r_1,r_2}$*

*Moreover for any file f, $(p, f, contain) \in Deps_{r_2}$ if exactly the same conditions are met. Since these conditions can be derived from
$SSn_{r_1}, Files^{add}_{r_1,r_2}, Files^{rem}_{r_1,r_2}, SSn^{\Delta}_{r_1,r_2}$, we can conclude proof of claim.* ■

Claim 1.3. *$Call_{r_2}$ and Ref_{r_2} can be determined on the basis of
$SSn_{r_1}, Files^{add}_{r_1,r_2}, Files^{rem}_{r_1,r_2}, SSn^{\Delta}_{r_1,r_2}$*

Proof of Claim 1.3. *According to the description from section 6.2.1, if:*

1. *m denotes a method with name* methodName,

2. *defined in entity e_2 within source code file f_2 (i.e. $(m, e_2) \in Cont_{r_2}$ and $(m, f_2) \in Cont^*_{r_2}$)*

3. *e_1 is an entity defined in file f_1 (i.e. $(e_1, f_1) \in Cont^*_{r_2}$)*

then $(e_1, m, call) \in Deps_{r_2}$ iff the source code of entity e_1 at revision r_2 contains a Java language expression in form:

$\varepsilon.methodName(\ldots)$

Where the type of expression ε corresponds to the type represented by e_2.

The expression ε must have the form : $\varepsilon_1.\varepsilon_2.(\ldots).\varepsilon_n$, *where each ε_i corresponds to either a class, a variable, a field or a method. Where applicable, we will equate ε_i with the respective entity for the sake of simplicity. Such expressions are evaluated during the traversal of the abstract syntax tree of the body of e_1, which is part of traversal of the abstract syntax tree of f_1, as described in section 6.2.1.*

If there is such i that the type of ε_i is different at revision r_2 than it was in r_1, it means that the source code of a file f_{ε_i} in which ε_i was declared at revision r_1 (i.e. $(\varepsilon_i, f_{\varepsilon_i}) \in Cont^*_{r_1}$) must be modified (i.e. $f_{\varepsilon_i} \in Files^{mod}_{r_1,r_2}$) or there is a class c_{ε_i} in which ε_i is directly contained (i.e. $(\varepsilon_i, c_{\varepsilon_i}) \in Cont_{r_1}$).

In both cases $f_1 \in Files^{aff}_{r_1,r_2}$, according to Definition 20. In such case, $(e_1, m) \in Call_{r_2}$ iff $(e_1, m) \in Call^{\Delta}_{r_1,r_2}$

If the type of all ε_i is identical at revision r_1 and r_2, then the abstract syntax tree traversal of such expression at r_2 must produce exactly the same elements of $Call_{r_2}$ as it produces at revision r_1 when $Call_{r_1}$ is produced. Consequently, in such case $(e_1, m) \in Call_{r_2}$ iff $(e_1, m) \in Call_{r_1}$

Clearly, when $f_1 \in Files^{add}_{r_1,r_2}$, then $(e_1, m) \in Call_{r_2}$ iff $(e_1, m) \in Call^{\Delta}_{r_1,r_2}$.

Therefore, we can conclude:

- *if $f_1 \notin Files^{\Delta}_{r_1,r_2}$ then $(e_1, m) \in Call_{r_2}$ iff $(e_1, m) \in Call_{r_1}$*
- *if $f_1 \in Files^{\Delta}_{r_1,r_2}$ then $(e_1, m) \in Call_{r_2}$ iff $(e_1, m) \in Call^{\Delta}_{r_1,r_2}$*

Since these conditions can be computed from $SSn_{r_1}, Files^{add}_{r_1,r_2}, Files^{rem}_{r_1,r_2}, SSn^{\Delta}_{r_1,r_2}$, so can the entire $Call_{r_2}$.

The reasoning about Ref_{r_2} is analogous, since the definition of affected file *and the procedure of traversing the abstract syntax tree during the evaluation of FP function is symmetric with respect to* call *and* refer *dependency types. Therefore, without repeating a rephrased proof from the above, we can state that Ref_{r_2} can be derived from $SSn_{r_1}, Files^{add}_{r_1,r_2}, Files^{rem}_{r_1,r_2}, SSn^{\Delta}_{r_1,r_2}$.* ■

Claim 1.4. *$Metr_{r_2}$ can be determined from $SSn_{r_1}, Files^{add}_{r_1,r_2}, Files^{rem}_{r_1,r_2}, SSn^{\Delta}_{r_1,r_2}$.*

Proof of Claim 1.4. *For a file $f \in Files_{r_2}$: If $f \notin Files^{add}_{r_1,r_2} \cup Files^{mod}_{r_1,r_2} \setminus Files^{rem}_{r_1,r_2}$ then the contents of f in r_2 are the same as they were in r_1. Since the metric can be computed from the source code only*[4]*, therefore for any metric m $Metr_{r_2}(e, m) = Metr_{r_1}(e, m)$ for any e such that $(e, f) \in Cont^*_{r_1}$*

In the opposite case $f \in Files^{\Delta}_{r_1,r_2}$ holds. In such case for any metric m $Metr_{r_2}(e, m) = Metr^{\Delta}_{r_1,r_2}(e, m)$ for any e such that $(e, f) \in Cont^*_{r_2}$. Since

[4]Please note a special case of *FI* and *DIH* metrics described in Section 6.2.1.

both conditions are encoded in $SSn_{r_1}, Files^{add}_{r_1,r_2}, Files^{rem}_{r_1,r_2}, SSn^{\Delta}_{r_1,r_2}$*,* $Metr_{r_2}$ *can also be derived from it.* ■

The proof of Theorem 1 is a direct consequence of Claims 1.1, 1.2, 1.3 and 1.4.

□

Theorem 1 has significant consequences in practical applications of the model proposed in this thesis. It is related to the efficient and adaptive construction of software evolution. The statement "without the use of $Source_{r_2}$ function" is just a formal way of saying that we do not need to parse the entire source code at revision r_2. It is sufficient to parse only files affected by this commit. Conceptually, it means that building a software snapshot is relatively inexpensive if the software snapshot that preceded it in the software evolution has been built previously. The following list describes sufficient steps to build $Files^{aff}_{r_1,r_2}$, once SSn_{r_1} is known:

- Take $Files^{mod}_{r_1,r_2}$ and $Files^{rem}_{r_1,r_2}$ from the logs of the SCM,
- Find all vertices of SSn_{r_1} that are reachable from elements of $Files^{mod}$ by following edges only from reversed $Cont_{r_1}$ relation. Store all visited nodes in Ent^{aff}. The set contains all software entities that are transitively contained in modified files.
- Find all nodes in SSn_{r_1} that are reachable from Ent^{aff} by following a path that starts with a positive number of edges with label *extend*, and ends with one edge *contain*. These are members of classes that inherit from classes already present in Ent^{aff}. Add all such elements to Ent^{aff}.
- Find all vertices of SSn_{r_1} that are reachable from elements of Ent^{aff} by following edges only from reversed $Call_{r_1} \cup Ref_{r_1}$ relation. Store all visited nodes in Ent^{call}. The set contains all software entities that are calling or referring at least one entity from Ent^{aff}.
- Find all vertices of SSn_{r_1} that are reachable from elements of Ent^{call} by following edges only from $Cont_{r_1}$ relation. Store all visited nodes that represent files in F. The set contains only files that contain a call or a reference to entities from Ent^{call}, which, by definition, is $Files^{aff}$, therefore $Files^{aff}_{r_1,r_2} = F$.

The number of visited nodes in such an algorithm is theoretically large, but in practice it rarely exceeds 5-10% of the total number of source files, according to the results of the experiment A.2.2.

Once the set of affected files is known, we can compute the differential snapshot, according to Definition 22 and then build SSn_{r_2} according to the rules described in the proof of Theorem 1. This yields an effective method of adaptive construction of $(SSn_r)_r$ along the development of the analyzed system.

6.3 Mining static software patterns

In this section we conceptually explain and formally define the problem of finding *software pattern instances*.

6.3.1 Software pattern instance

Please recall from Section 2.2.8 that design patterns and design anti-patterns are certain conceptual structures in the software. Given a software snapshot, we can semi-formally say that they are certain subgraphs.

In the following deliberations, we discern two notions considered in the context of SSn: the *pattern* and the *pattern instance*. The former is an arbitrary subset of subgraphs of SSn corresponding to the concept described by (frequently informal) specification, whereas the latter is a specific subgraph SSn. For example, the pattern is a set defined by the concept '*a class with cyclomatic complexity over 100, calls a method with less than 2 lines of code*', whereas its instance in SSn is each subgraph $PI = (V_{PI}, E_{PI}, Metr_{PI})$ of SSn such that:

1. $V_{PI} = \{c, m\}$ where c, m are class and method respectively.
2. $Metr(c, Cyclomatic) > 100$ and $Metr(m, NCSS) < 2$
3. $(c, m, call) \in E$,
4. E_{PI} and $Metr_{PI}$ are respective subsets of E and $Metr$ induced by V_{PI}.

Formally, the pattern is an arbitrary subset of the set of subgraphs of SSn and each of its elements is the pattern's instance[5].

[5]We will later slightly extend the notion of pattern instance with the *main entity*

Pattern instance identity

In Section 6.1.1 we have defined the function $Source_?$ to discern software entity and its source code, so that we can observe the changes of the entity over time. Similarly, we want to be able to observe changes of instances of software patterns, therefore each instance must be uniquely identified over time.

Main entity of pattern instance

In order to uniquely define the identity of the pattern instance we will introduce the notion of the *main entity*, which is one selected node that can be specifically appointed in the subgraph. This is motivated by the fact that the definitions of software anti-patterns are hardly ever symmetric and only one entity plays in it a crucial role. For example, an informal definition of Blob: "A class with too many complex methods", refers to more than one entity (i.e. class, methods contained in it) but clearly the entity corresponding to the class is the main element of this pattern. Therefore, the formal representation of each instance of this pattern is a subgraph, which contains one entity with the class (c), the number of complex methods ($\{m_i\}_i$) contained in c. Intuitively, in such a graph it is c that will be considered the main entity. A more formal definition of the main entity can be found in Section 6.3.2. If a pattern instance has a main entity, then the identity of this instance is given by the type of pattern and the name of the main entity. Please note that even if the structure of the corresponding subgraph changes over revisions, such identity may remain intact.

However, some types of static patterns, (e.g. circular dependency), do not have a single selected entity. In such cases we assume that the identity of the pattern instance is given by the set of names of all entities contained in the corresponding subgraph.

6.3.2 Software pattern mining formalism

Some previous examples of a *pattern* are crisp and all its instances can be precisely found in any software snapshot by a trivial search. Yet, actual design anti-patterns are usually vague. A good example is the *Swiss army knife* (see [343]) which is a 'class with many complex unrelated functionalities'. In

described in Section 6.3.1 and containment-completeness from Definition 23.

this case there is no straightforward method of translating such an informal concept into graph-theoretical terms.

Definition 23 (containment-completeness)**.** *We will say that subgraph* $g = (V_g, E_g)$ *of* SSn *is* containment-complete*, if for any* $e_1 \in V_g$ *if there is a node* $e_2 \in V$ *such that* e_2 *is contained in* e_1 *in* SSn *then* $e_2 \in V_g$*.*

Since the structure of software is represented as a graph, it is natural to assume that pattern instances are simply specific subgraphs, as explained in preceding paragraphs. For practical reasons we will assume that such graph must be containment-complete, which is a natural way of looking at the structure of software: If we extract a fragment of the structure source code, we also want all its sub-structures to be extracted for completeness (e.g. if we extract a single class as a part, we want also the methods and fields of this class to be extracted). Therefore, unless stated otherwise, in the following paragraphs a subgraph always means a containment-complete subgraph.

Formalism for mining static patterns

Let $\mathscr{P}(SSn)$ denote the set of all containment-complete subgraphs of a graph SSn. In the context of SSn, the pattern P can be any subset of $P \subseteq \mathscr{P}(SSn)$, and the pattern instance is each subgraph $g \in P$. The problem of mining static software patterns can be therefore defined as follows:

Formally, we can say that the problem of mining software pattern S, is to find such a deterministic program P_S (called *classifier*), which, given any $SSn \in \mathcal{SSn}$, outputs the set of instances of the pattern in SSn, provided that the program can query its input only for the following properties of the SSn graph:

- the set of nodes, edges and their types,
- the set of edges adjacent to the given node,
- the set of nodes adjacent to the given edge,
- the value of the $Metr$ function for any input.

Additionally, we will assume that the classifier must be invariant to the change of the names of the entities, just like software metrics (see Section 2.2.7) are. This is a simplifying assumption, since some researchers (e.g. [243])

also use lexical properties that are derived from the names of the software entities. Yet, as empirically shown in the experiments in Appendix A.5.1, the simplified model presented in this thesis can be comparable and sometimes even better in terms of classification quality, when compared to the state of the art software pattern mining algorithms.

In contexts where the identity of a pattern instance is needed, we will also assume that the classifier also outputs the main entity of each returned subgraph, if applicable. Formally, we will expect that $P_S(SSn) = \{(P_1, e_1), \ldots, (P_m, e_m)\}$, where $\{P_1, \ldots, P_m\}$ forms a complete set of instances of pattern S in SSn, and for each i e_i is a *main entity* of P_i, if P_i has one or $\perp$ otherwise. Yet, we omit this aspect in other contexts, for the sake of simplicity.

Suppose that for given software we have a list of all instances of a certain software pattern, e.g. given by an expert. If we parse the source code of this software to build a corresponding software snapshot SSn, the list maps naturally to certain subset S of subgraphs of SSn. Ideally, the classifier P_S should be able to output exactly S. In such case we will say that S is *defined* by P_S. In practice, this is not always the case, and the difference between S and $P_S(SSn)$ allows us to determine the quality of classification of P_S e.g. by means of measures described in Section 4.1.4.

An equivalent definition is that the classifier can be constructed on the basis of the characteristic function of the output set. Suppose that the program P'_S can accept any element of $\mathscr{P}(SSn)$ as an input, it can query the SSn graph according to the rules described above and it outputs either 1 or 0 according to the following rule:

$$P'_S(x) = \begin{cases} 1 & \text{if } x \in S \\ 0 & \text{otherwise.} \end{cases}$$

In such case we will also say that P'_S *defines* S, and for any x such that $P'_S(x) = 1$ we will say that P'_S *finds* x. In the following paragraphs we use both definitions of the classifier interchangeably.

Definable and indefinable patterns

Naturally, a question arises if all software patterns can be properly identified by a classifier program, given that it can only query the structure of the SSn graph according to the rules defined above. In the following paragraphs we will discuss this matter formally.

Definition 24. *Let $S \subseteq \mathscr{P}(SSn)$ be a pattern. We will say that S is* definable *if there is a classifier P_S such that $P_S(SSn) = S$. In the opposite case S is* indefinable.

Fact 1. *Not all sets are definable.*

Listing 6.5 contains an example of software whose structure contains indefinable patterns.

Listing 6.5: exemplary software, whose structure contains indefinable patterns.

```
class A{}
class B{}
```

The corresponding software snapshot consists of just two isolated nodes - representing classes A and B respectively. Since software metrics are invariant of any rename of the software entities, the values of respective metrics for these two classes must be equal. Therefore, no classifier can discern A and B, which clearly means that singleton sets of subgraphs induced by either $\{A\}$ or $\{B\}$ are indefinable.

Fact 2. *The following sets are definable:*

- $\emptyset$
- $\mathscr{P}(SSn)$

Proof.
- The classifier that always outputs 0 defines $\emptyset$.
- The classifier that always outputs 1 defines $\mathscr{P}(SSn)$.

□

Fact 3. *If A and B are definable sets then all the following sets are also definable:*

- $-A$
- $A \cup B$
- $A \cap B$

Proof. Suppose that P'_A defines A and P'_B defines B. Therefore clearly:

- $P'_{-A}(x) \overset{def}{\equiv} 1 - P'_A(x)$ defines $-A$,
- $P'_{A\cup B}(x) \overset{def}{\equiv} max(P'_A(x), P'_B(x))$ defines $A \cup B$,
- $P'_{A\cap B}(x) \overset{def}{\equiv} min(P'_A(x), P'_B(x))$ defines $A \cap B$.

$\square$

Definition 25. *For any pattern S:*

Any definable set X such that $X \subseteq S$ is called the sub-approximation *of S.*

Any definable set X such that $S \subseteq X$ is called the super-approximation *of S.*

Theorem 2. *For any pattern $S \in \mathscr{P}(SSn)$ there are definable sets $\overline{S}$ and $\underline{S}$ such that:*

- $\underline{S} \subseteq S \subseteq \overline{S}$
- *There is no other definable set X such that $\underline{S} \subset X \subset S$*
- *There is no other definable set X such that $S \subset X \subset \overline{S}$*

Proof. Let $\overline{\mathscr{S}} = \{X : X$ is super-approximation of $X\}$ and $\underline{\mathscr{S}} = \{X : X$ is sub-approximation of $X\}$. Since SSn is finite, so $\overline{\mathscr{S}}$ and $\underline{\mathscr{S}}$ are. Therefore,

$sup(\underline{\mathscr{S}}) \overset{def}{\equiv} \bigcup_{x\in\underline{\mathscr{S}}} x$ and $inf(\overline{\mathscr{S}}) \overset{def}{\equiv} \bigcap_{x\in\overline{\mathscr{S}}} x$ are definable according to the Fact 3. Clearly, $sup(\underline{\mathscr{S}}) \subseteq S \subseteq inf(\overline{\mathscr{S}})$.

Suppose that there is definable X such that $sup(\underline{\mathscr{S}}) \subset X \subset S$. Since X is definable and $X \subset S$, $X \in \underline{\mathscr{S}}$. In such a case $X \subseteq \bigcup_{x\in\underline{\mathscr{S}}} = sup(\underline{\mathscr{S}})$, which is contrary to the assumption above. Consequently, such X does not exist.

By analogy, we can prove that there is no X such that $S \subset X \subset inf(\overline{\mathscr{S}})$

In the end we can put: $\underline{S} = sup(\underline{\mathscr{S}})$ and $\overline{S} = inf(\overline{\mathscr{S}})$

$\square$

$\underline{S}$ is the greatest sub-approximation and will be called the *lower approximation* of S and $\overline{S}$ is the least super-approximation and will be called the *upper approximation* of S. They conceptually represent the most accurate sub (respectively: super) approximation of pattern S that any classifier can potentially produce.

The difference $\overline{S} \setminus \underline{S}$ will be called the *boundary region* of S and it conceptually represents such subgraphs of SSn that cannot be either included in or excluded from to S by any classifier.

6.3.3 Graph-isomorphism-based mining

The above definitions of the mining problem refer to a general *program* which given any software snapshot identifies all instances of a certain software pattern. Arguably the most natural way of implementing it is based on the SUBGRAPH-ISOMORPHISM problem. Formally if $M = \{m_1, \ldots, m_n\}$ is a set of software metrics, $R_S = (V_{R_S}, E_{R_S}, Metr_S)$ is a *labeled reference graph*, where $Metr_S$ is a function[6] $Metr_S : V_{R_S} \times M \rightarrow P(\Re \cup \{\bot\})$ that provides the labeling of vertices in V_{R_S}, according to the convention defined in Section 6.1.1: i.e. each vertex $v \in V_{R_S}$ is labeled by a vector $(Metr_S(v, m_1), \ldots, Metr_S(v, m_n))$. For a given software snapshot $SSn = (Ent, Deps, Metr)$ we will say that a subgraph $s \in \mathscr{P}(SSn)$, whose set of vertices and edges is denoted V_s and E_s, *matches* R_S iff: (V_s, E_s) is isomorphic to (V_{R_S}, E_{R_S}) where the isomorphism is given by $i : V_{R_S} \rightarrow V_s$ and the following *metrics condition* holds: $\forall v \in V_{R_S}, m \in M \ Metr(i(v), m) \in Metr_S(v, m)$.

We will say that the set of all subgraphs of SSn that match R_S is *defined* by R_S. The corresponding program is straightforward: Given a SSn in $\mathcal{SSn}$ the program first computes the set of subgraphs isomorphic to (V_{R_S}, E_{R_S}), and then removes such elements of this set which do not satisfy the metrics condition. Such an approach is valid, yet very inefficient in practice, since in practical applications the number of nodes that satisfy the metrics condition in the complete SSn graph tends to be very low, in some cases not exceeding 1%. This observation has been experimentally validated in Experiment A.5.2. Therefore, a more efficient heuristic is to filter the entities by the metrics condition first and then to search for pattern instances only among specific, relatively small subgraphs, which contain such filtered nodes. Conceptually, it means that costly SUBGRAPH-ISOMORPHISM heuristics may be applied to only small fragments of the entire SSn graph. Moreover, usually, specific metric condition concerns the main entity of the anti-pattern (see Section 6.3.1), which in practice puts a constraint on the isomorphism function. Detailed description of the heuristics based on this concept is given

[6]Please note different symbols: $\mathscr{P}(X)$ denotes the set of containment-complete subgraphs of X, while $P(Y)$ denotes the power set of Y.

in section 6.4.

Section 6.2.3 describes a method for adaptive evaluation of software-evolution structure, which typically requires re-applying the costly FP function to a small fragment of source code files (see A.2 for experimental validation). This adaptive heuristic provides the SSn_{r_i} structure for the series of subsequent revisions (r_i). This fact, combined with the previous observation that metric filters allow to significantly reduce the cost of mining certain pattern instances, leads to the conclusion that there is a more efficient heuristic method for mining series of software pattern instances indexed by the revisions. Conceptually: at each revision it is sufficient to check if entities modified in this revision satisfy the metrics condition and only then apply SUBGRAPH-ISOMORPHISM algorithm on a small subgraph around them. This concept is further considered in Section 6.2.3. In the following sections we introduce a more general framework for defining and measuring the complexity of classifiers.

6.3.4 Complexity of pattern mining

We will measure the complexity of the classifier by the number of queries about the SSn graph it has to use in order to produce the output. The following definitions yield a formal concept that will allow us to provide some estimation on the complexity of classifiers described in this thesis. In the above formalism we have used the concept of a program that can query entire SSn in order to determine if a given, potentially very small subgraph is an instance of a pattern. Intuitively and practically, such a program can probably query only a bounded neighborhood of a subgraph given as input to be able to produce a deterministic output. The following definitions provide some formal constructs that will enable us to estimate how small such neighborhood can be:

Definition 26 (d-reachable nodes)**.** *Let $V_s \subseteq V$ be a sub-set of nodes from graph $SSn = (V, E)$ and $d = (d_i)_{i=1\ldots n}$ be a finite sequence of labels from Dep_Types. d-reachable nodes from s, is a set of nodes $(V_s \xrightarrow{d})$ such that:*

$x \in (V_s \xrightarrow{d})$ iff at least one of the following conditions holds:

- *$x \in V_s$*
- *There is $k \in \{1, \ldots, n+1\}$ such that SSn contains a path $(v_1, \ldots v_k)$ where $v_1 \in V_s$ and for each $1 < j \leqslant k$ SSn contains edge (v_{j-1}, v_j) or*

edge (v_j, v_{j-1}) *of type* d_{j-1}

Intuitively, nodes d-reachable from V_s contain only the vertices that can be reached by a path that starts at a node from s and follows edges (in either direction) according to the sequence of labels from d. In the following sections we will also use a dual term:

Definition 27 (reverse d-reachable nodes)**.** *Given symbols as in Definition 26* reverse d-reachable nodes *from* V_s *is* $(V_s \xleftarrow{d}) \overset{def}{=} \{x$ *: there exists a node* $e \in V_s$ *such that* e *is d-reachable from* $x\}$*.*

These two notions are explained by Figure 6.1.

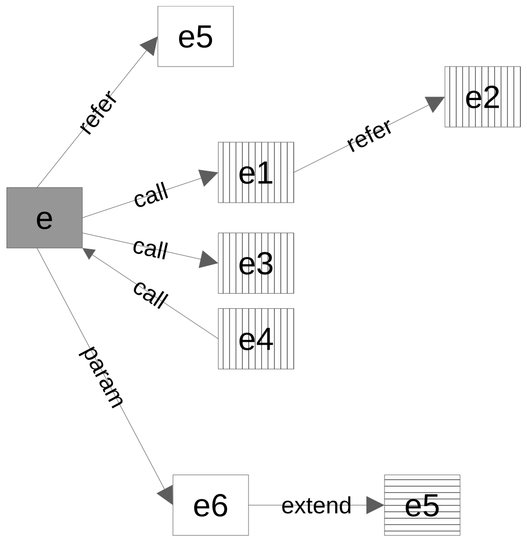

Figure 6.1: In this exemplary graph nodes with vertical stripes ($e1$-$e4$) are ($call$, $refer$)-reachable from node e. Node $e5$ is ($extend$, $param$)-reverse-reachable from node e.

Definition 28 (D-surrounding of a graph)**.** *Let* $s = (V_s, E_s) \in \mathscr{P}(SSn)$ *be a subgraph of* $SSn = (V, E)$ *and* $D = \{D_1, \ldots, D_k\}$ *be a finite set of sequences of labels from Dep_Types, as given in definition 26.* A D-surrounding of s *is a subgraph of* SSn *induced by set of nodes* $\bigcup_{i=\{1,\ldots,k\}}(V_s \xrightarrow{D_k})$

Definition 29 (reverse D-surrounding of a graph)**.** *Given symbols as in Definition 28,* reverse D-surrounding of graph s *is a subgraph induced by nodes that are reverse* D_i *reachable from a node from* V_s *for any* $i \in 1, \ldots, k$*.*

Definition 30 (D-bounded classifier). *Let D be defined as in Definition 28 and P be a classifier. We will say that P is* D-bounded *if it satisfies the following conditions:*

- *for any $SSn \in \mathcal{SSn}$ and for any $s \in \mathscr{P}(SSn)$, P only queries about nodes or edges that are contained in the D-surrounding of s to produce the output.*
- *if s is the pattern instance and e is its main entity then P queries only about nodes or edges that are contained in the D-surrounding of graph induced by $\{e\}$ to produce the output.*

The following fact is a straightforward consequence of Fact 3 and Definition 30:

Fact 4. *Let P_A be a D_A-bounded classifier that defines A and Let P_B be a D_B-bounded classifier that defines B. In such case there exists a $(D_A \cup D_B)$-bounded classifier for $A \cup B$ and $A \cap B$.*

Section 6.4 contains strict definitions of classifiers for a few popular design anti-patterns, followed by observations that they are D-bounded with some relatively compact D. In the following sections we argue that there are efficient heuristics to find entities which may be the *main entities* of a pattern instance and we show that these are infrequent. Next, we combine the concept of D-bounded classifier to produce a heuristically-efficient classifier for popular design anti-patterns in such a way that it checks for the existence of the instance of a pattern only within a D-surrounding of each *main entity*-candidate.

6.3.5 Adaptive mining of pattern instances

In Section 6.2.3 we have shown that the construction of a following software snapshot in software evolution can be implemented adaptively and thus leverage the fact that typically only very limited fragment of the system source code changes in two subsequent revisions (see Appendix A.2). Similar construct also applies to mining the instances of software patterns. Suppose that r_1 and r_2 are two subsequent revisions and the set of instances of software patterns at revision r_i is denoted by $Inst_{r_i}$. Let C be a D-bounded classifier for some known D, as described in Definition 30. Moreover, let F_{r_i}

be a sub-set of entities at revision r_i with such a property that if a pattern instance PI at revision r_i matches C then the main entity of PI is in F_{r_i}.

Figure 6.2 shows the conceptual procedure of finding $Inst_{r_2}$, when $Inst_{r_1}$ (r_1 precedes r_2) is known: At first, SCM logs are analyzed to find the set of files affected by commit r_2 (arrow 1) and then to find entities affected by revision r_2 (arrow 2). Please recall that these two steps, described in Section 6.2.3 are part of regular adaptive construction of software evolution presented therein. Experiments described in Appendix A.2 show that the number of affected entities is typically relatively small.
In the next step (arrow 3) the set of all entities that are reverse *D-reachable* from at least one affected entity is computed, which yields the first approximation of the set of candidates for main entities of pattern instances defined by C. This set will be denoted $Cand_1$. Experiment A.4.1 shows that the number of such entities tends to be relatively small.
In the following step (arrow 4) all such candidate-entities are further filtered according to the *metrics filter* defined in Section 6.4(technically we will represent this filtering by intersection with F_{r_2} set), which produces a final set of candidates for the main entities ($Cand_2$). In this case experiment A.5.2 shows that again the number of such entities is typically very small. In short, the $Cand_2$ set is the set of entities D-reachable from entities reverse D-reachable from entities affected by revision r_2, which satisfy the metric condition for C. At the end, the classifier C is applied on the subgraph induced by $Cand_2$.

Theorem 3. *Let r_1, r_2, C, D, F_{r_2}, $Cand_1$, $Cand_2$, $Inst_{r_1}$ and $Inst_{r_2}$ be defined as above. If a pattern instance PI of type defined by C with main entity is present in revision r_2, at least one of the conditions holds:*

- *PI was present in r_1 and none of the nodes of PI is reverse reachable from any entity affected by r_2.*
- *C finds PI in the subgraph induced by $Cand_2$.*

Proof. Evaluation of $C(PI)$ queries SSn_{r_2} about nodes that are *D-reachable* from some node of PI and edges adjacent to such nodes. If none of the nodes of PI is reverse *D-reachable* from any entity affected by r_2, it will never query about any entity affected by r_2 or its adjacent edges. Therefore, all queries of C produce the same result in r_1, which means that PI was present in r_1.

If pattern PI was not present in r_1 and is present in r_2, it means that its main entity $m \in F_{r_2}$ must be reverse *D-reachable* from nodes affected by r_2, because C is *D-bounded*. Moreover, P will only query D-surrounding of m, which is contained in $Cand_2$, in order to produce a deterministic output.

If pattern PI was present in r_1 and is not present in r_2 it means that its main entity m must be reverse D-reachable in r_1 from some entity that is affected by r_2. If m is present in r_2 and $m \in F_{r_2}$ then C will query only nodes *D-reachable* from m (and their adjacent edges), which are all contained in $Cand_2$. Consequently, C allowed to query $Cand_2$ will not find it. If m is not present in r_2 or $m \notin F_{r_2}$ then clearly C will not find PI in r_2. $\square$

Theorem 3 yields a straightforward method for adaptive mining of pattern instances in software evolution: In order to produce $Inst_{r_2}$ from $Inst_{r_1}$ it is sufficient to subtract pattern instances with at least one entity reverse D-reachable from entities affected by r_2 and then add all instances found by C on the subgraph induced by $Cand_2$. As mentioned earlier in this section, experiments show that typically these operations need to be performed on very small subgraphs (precisely: on subgraphs built from a very small subsets of nodes).

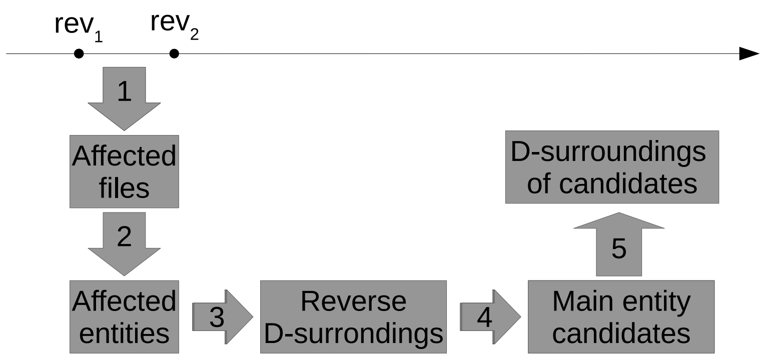

Figure 6.2: The concept of adaptive mining of pattern instances.

6.4 Definable patterns for anti-patterns and code smells

In this Section we present a set of heuristic methods of finding instances of a few popular design anti-patterns which are based on the model described earlier in this chapter (see Sections 6.3.2 - 6.3.5). The formal method for measuring the complexity of these algorithms is described in Section 6.3.5 and conceptually is based on the fact that only a limited fragment of the entire *SSn* graph must be visited, along specific paths, starting at specific nodes. Each of the following Subsections 6.4.1-6.4.7 describes a method for finding instances of a single type of design anti-pattern and its complexity in the aforementioned terms. Each method is derived from available studies from the related work mentioned in the respective subsection. It has been adopted to the software snapshot model described in this thesis and improved, where applicable. The specific constants in each definition (e.g. thresholds for source code metrics) were taken from original work or manually calibrated to fit to the reference data described in Section 6.7.1 in terms of F1.

For greater clarity, the algorithms for finding the instances of static anti-patterns in the following subsections are explained in terms of graphs, software entities, software engineering and the Java programming language. Yet, all these descriptions can be formalized according to the rules described in Section 6.3.2. In most cases the description refers to a single class which clearly is the main entity of the respective pattern.

6.4.1 Anemic entities

Anemic entities (sometimes called data classes) are classes which only store data and do not provide any functionality (see [121], [287]). A straightforward, naive approach for detecting this pattern is to take classes which have: many fields, only accessor methods (i.e. methods with single line of code, which refer to only a single field and have cyclomatic and npath complexity equal to one), default and possibly an initializing constructor. This heuristic turns out be inaccurate, since experimental validation (see Appendix A.5.1) shows that there are anemic entities which have methods that *effectively* should be considered accessors but do not satisfy the above condition (e.g. they contain some validation or initialization logic) or a constructor with excessive logic. Therefore, we need to define two notions: an *effectively trivial method* and *complex constructor*.

A method m is *effectively trivial* if :

1. The class c in which m is contained has field f of type t such that m refers to f and either m has 1 argument of type t and *void* return type or it has 0 arguments and return type t,

2. m has at most 5 lines of code,

3. m has cyclomatic complexity not greater than 3.

The constructor *con* contained in c is *complex* if:

1. The number of arguments of *con* exceeds the number of fields contained in c,

2. the lines of code in *con* exceeds 150% of the number of fields contained in c,

3. the cyclomatic complexity of *con* exceeds 150% of the number of fields contained in c.

This allows us to provide a definable pattern for anemic entity: A class is an anemic entity iff:

1. it has more than 8 fields,

2. it has more than 8 methods,

3. all methods but one are trivial or *effectively trivial*,

4. there are no complex constructors contained in c,

5. all subclasses of c satisfy the above four conditions.

A classifier that does not check the 5 condition is clearly $\{(contain)\}$-*bounded*. Since the arguments must be checked for all classes in the hierarchy of an examined class entity, we can conclude with the following fact:

Fact 5. *Let H be the maximum depth of inheritance tree of any class in the entire evolution of the software, and let $D_H = \{(contain)\ , (extend, contain)\ , \ldots, (\overbrace{extend, \ldots, extend}^{H\ times}, contain)\ \}$. The above classifier for anemic-entity is D_H-bounded.*

The maximum depth of the inheritance tree H tends to be low and in the analyzed systems does not exceed 7. Therefore the proposed classifier for Anemic entity is efficient. Both these statements are experimentally validated by experiments described in Appendix A.4.3.

6.4.2 Swiss army knife

A *Swiss Army Knife* (*SAK* for short), is an excessively complex class interface. It is present when e.g. the creator attempts to provide a method for all possible uses of the class or make a single class serve many complex unrelated functions. Methods described or referenced in [241], [244], [342] and [243] provide a semi-formal description of the pattern as *a class with many unrelated methods with high complexity which implements many interfaces*. Clearly, this can be translated to the graph-theoretical language in the context of a software snapshot: a class with many interfaces corresponds to each node such that paths which start from it and contain edges of type *extend* and *implement* reach many nodes which represent interface entity. A complex method is simply a method with high values of the complexity metrics such as *NPath complexity* or *cyclomatic complexity.* Additionally, if a class is intended to serve many purposes, one can expect that it has methods that are being called by many other classes. This conceptual description can be translated in the following formal definition of a definable pattern which approximates the SAK anti-pattern:

Definition 31 (foreign call)**.** *For a given entity e a* foreign call *is each edge* $(caller, callee)$ *such that:*

1. $(caller, callee) \in call$,
2. $(e, callee) \in Cont$,
3. *there are no nodes* sup_1, sup_2 *such that* $(sup_1, e) \in Cont^*$ *or* $e = sup_1$ *and* $(sup_2, caller) \in Cont^*$ *and* $\{(sup_1, sup_2), (sup_2, sup_1)\} \cap Ext^* \neq \emptyset$.

Conceptually, a foreign call is a call of a method from a member of another class hierarchy.

Swiss Army Knife is a class c that satisfies the following conditions:

1. c has more than 6 methods,
2. the value of metric LOC for c exceeds 150,
3. the sum of cyclomatic complexity of methods contained in c exceeds 30,
4. the sum of NPath complexity of methods contained in c exceeds 120,

5. the number of effectively non-trivial methods contained in c multiplied by the average NPath complexity of such methods exceeds 160,

6. the number of methods called by a foreign call exceeds 2,

7. the number of foreign calls exceeds 7.

In order to say if a given class c is a main entity of an instance of SAK, it is sufficient to visit all its superclasses and subclasses, all their methods and calls of these methods. In terms of SSn graph traversal, if we know that the maximum depth of an inheritance tree in the entire evolution of a system is bounded by some constant H, then we may define a bounded set of sequences of edges in the following manner: $D_H = \{(contain, call, contain),$ $(extend, contain, call, contain)\ , \ldots,$
$(\overbrace{extend, \ldots, extend}^{H \text{ times}}, contain, call, contain)\ \}$

We may now conclude with the following fact:

Fact 6. *The above classifier for the Swiss Army Knife is D_H-bounded*

6.4.3 God class and Brain class

God class (also called *Blob*) and *Brain class* are similar anti-patterns that refer to classes that provide too much complexity and tend to centralize logic of some area of the system. The difference between God class and Brain class is that the former is a large controller class that depends on data from surrounding classes whereas the latter does not use the data from other classes, tends to be more cohesive and encapsulates the logic in its own complex methods (see [228], [258], [342], [174]). Consequently, a God class is a class that: 1. has many complex methods and fields 2. does not use data from other classes, i.e. it neither refers to fields of other classes nor calls trivial methods of other classes 3. tends to call its own methods.

A Brain class is a class that: 1. has many complex methods and fields 2. uses data from other classes, i.e. refers to fields of other classes or calls accessor methods of other classes 3. tends to call its own methods 4. tends to have 'controller' methods: either just a few complex ones or more moderately complex ones. These informal concepts can be translated in the following definable patterns:

God class

Entity c is considered a god class iff:

1. c calls more than 7 effectively trivial methods of other classes,

2. proportion of the number of effectively trivial methods of other classes called by c to the number of other methods of other classes called by c exceeds 0.6,

3. the sum of cyclomatic complexity of its non-trivial methods exceeds 55,

4. the value of the metric *tight class cohesion* does not exceed 0.3.

Clearly, to determine if a given class c satisfies the above definition, it is sufficient to traverse SSn along all paths that start at c and have consequent edge labels *contain*, *call*, $refer$. Therefore, we can conclude with the following fact:

Fact 7. *This classifier for God class is* $\{(contain, call, refer)\}$*-bounded.*

Brain class

Entity c is considered brain class iff

1. c is not god class, as defined in the preceding subsection,

2. c has more than 2 non-trivial methods with more than 4 outgoing edges of type *call* and more than 15 lines of code (controller methods),

3. c calls more than 5 trivial methods of other classes,

4. proportion of the number of trivial methods of other classes called by c to the number of non-trivial methods of other classes called by c does not exceed 0.6,

5. the number of calls to trivial methods divided by the number of lines of the source of c is smaller than 0.2,

6. the value of tight class cohesion metric for c does not exceed 0.5,

7. One of the following conditions is true: (a) sum of cyclomatic complexity of methods contained in c exceeds 50 and the value of NCSS metric for c exceeds 400 or (b) sum of cyclomatic complexity of methods contained in c exceeds 90 and the value of NCSS metric for c exceeds 50.

By analogy to the god class classifier, on the basis of Fact 4, we can conclude with the following fact:

Fact 8. *This classifier for Brain class is* $(contain, call, refer)$*-bounded.*

6.4.4 Base bean

Base bean is a class which only provides utility methods for its subclasses.

A method m contained in class c is an *utility method* iff:

- m is neither a constructor nor a trivial method,
- m does not refer any field contained in c,
- there is no s such that $(c, s) \in \mathit{Ext}^*$ and m references a field contained in s,
- there are no such s, f, cal such that $s = c$ or $(c, s) \in \mathit{Ext}^*$ and $(s, f) \in \mathit{Cont}$ and $(\mathit{cal}, f) \in \mathit{Ref}$ and $(m, \mathit{cal}) \in \mathit{call}$,
- the return type of m is not *void*,
- if $(\mathit{cal}, m) \in \mathit{call}$ then there is s such that $(s, c) \in \mathit{Ext}^*$ and $(s, \mathit{cal}) \in \mathit{Cont}$.

Conceptually, a utility method is a non-trivial method that does not modify state or orchestrate other methods of the class it is defined in or any of its ancestors and is used only by its descendants.

A class c is *base bean* iff:

- it has more than 2 utility methods,
- the number of descendants of c (i.e. the power of $\{e : (e, c) \in \mathit{Ext}^*\}$) is at least 5,

- the number of incoming edges of type *call* to utility methods contained in c exceeds 2.

Clearly, to tell if a class c is an instance of a *Base Bean*, it is sufficient to query about 1. methods and fields contained in c, 2. return types of methods contained in c 3. methods contained in all subclasses and superclasses of c, 4. fields referenced by all aforementioned methods 5. callers and callees of all aforementioned methods.

If we assume that H denotes the maximum depth of an inheritance tree in the entire evolution of the system, then we can construct the following finite set

$D_H = \{(contain, call)\} \cup \{(contain, type)\} \cup \{(extend, contain, refer)\}$
$\cup \{(extend, contain, call)\} \ldots \cup \{(\overbrace{extend, \ldots, extend}^{H\text{times}}, contain, refer)\}$
$\cup \{(\overbrace{extend, \ldots, extend}^{H\text{times}}, contain, call)\}$
and conclude with the following fact:

Fact 9. *The above classifier for base bean is D_H-bounded*

6.4.5 Yo-yo

A *YoYo* is a pattern where the flow control is scattered over overcomplicated inheritance structure, so that in order to understand the algorithm in the source code, one has to switch between many classes within a common inheritance tree (see [319], [316], [342]). This highly informal definition can be rephrased to graph-theoretical terms: There are many edges of type *call* or $refer$ between nodes which are directly contained in nodes that form a tree from edges of type *extend*. A precise definable pattern for YoYo is: A subgraph induced by nodes $Y = \{e_1, \ldots, e_n\}$ is a *YoYo* with main entity e_1 iff:

- Each pair $(e_i, e_j) \in Y \times Y$ is connected by a path constructed from edges of type *extend*, where each edge is treated as undirected,
- the longest path between any two nodes from Y constructed from such edges exceeds 5,
- The number of edges (m_1, m_2) of type *call* such that m_2 is not a trivial method and $\{(e_i, m_1), (e_j, m_2)\} \subseteq Cont$, $i \neq j$ exceeds 5.

- there is no super-set of nodes $Y' \supseteq Y$ such that graph induced by Y' satisfies the above three conditions.

The main entity e_1 of this graph is the class that does not have a superclass (the above conditions guarantee that there is always exactly one such class).

In order to determine if a given class is the main entity of an instance of a YoYo Pattern, it is sufficient to visit all classes in its inheritance hierarchy (*extend* edge), methods contained in all these classes (*contain* edges) and calls of these methods (*call* edges). Therefore we can conclude with the following fact:

Fact 10. *Let H be the maximum depth of any inheritance tree in the entire software evolution and let D_H be defined as follows: $D_H = \{(contain, call)\} \cup \{(extend, contain, call)\} \cup \ldots \cup \{(\overbrace{extend, \ldots, extend}^{H\,times}, contain, call)\}$. The above classifier for YoYo is D_H-Bounded*

6.4.6 Data Clumps

A *Data Clump* is code smell that occurs when a group of data items are being passed together in the source code (see [122], [268], [266], [361]). This informal definition can be rephrased in software engineering terms as: a group of methods, with substantial portion of identical parameters, such that any two are connected by an invocation chain. The corresponding definable pattern is:

Let $parameters(m) = \{p \in Ent : (m, p) \in Params\}$ A set of method entities $M = \{m_1, \ldots, m_n\}$ is a *Data Clump* iff:

- n exceeds 3,
- for at most one m_i $LOC(m_i) \geqslant 5$,
- $|parameters(m_i)|$ exceeds 3 for each i
- for each pair (m_i, m_j) such that $i \neq j$ *and* $(m_i, m_j) \in call$, $|parameters(m_i) \cap parameters(m_j)| = min(|parameters(m_i)|, |parameters(m_j)|)$
- there is no such superset of M that satisfies the above conditions

In order to tell if a given method m is a part of *Data Clump* it is sufficient to traverse paths that start at m and have edges with label *call* and to visit all nodes that are connected via *parameter* edge to one of the nodes on traversed paths. Therefore, we can conclude with the following fact:

Fact 11. *Let* call path *between methods* (m_1, m_2) *be a path without cycles constructed from method nodes with edges labeled only by call, so that no shorter path with the same property connects* m_1 *and* m_2*. Let* C *denote the maximum length of any call path in the entire evolution and let* D_C *be defined as follows:* $D_C = \{(parameter)\} \cup \{(call, parameter)\} \cup \ldots \cup \{(\overbrace{call, \ldots, call}^{C times}, parameter)\}$

The above classifier for Data Clumps *is* D_C*-bounded.*

Experiment A.4.3 shows that the constant C does not exceed 12, which makes the proposed classifier efficient.

6.4.7 Circular dependency

Circular dependency is a relation between two or more software entities transitively contained in different unrelated packages which either call each other directly or indirectly to function properly. We can translate this into graph theoretical terms:

A pair of classes (c_1, c_2) form a circular dependency iff:

- there exist two different packages p_1, p_2 such that $\{(c_1, p_1), (c_2, p_2)\} \subseteq Cont^*$ and neither p_1 is a subpackage of p_2 nor p_2 is a subpackage of p_1,
- there are two methods m_1, m_2, such that $\{(m_1, c_1), (m_2, c_2)\} \in Cont$,
- there is a path build from edges labeled *call* from m_1 to m_2 and another such path from m_2 to m_1.

In order to tell if two classes form a circular dependency it is sufficient to check what package they are transitively contained in, find all methods contained in them and then find call paths of these methods. We can therefore conclude with the following fact:

Fact 12. *Let* call path *be defined as in Fact 11 and Let* C *denote the maximum length of a call path in the entire evolution and let* T *denote the*

maximum length of a path with "contain" edges in the entire evolution. Let D_T *be defined as follows:* $D_T = \{(\overbrace{contain, \ldots, contain}^{T times}\}$
$\cup \{(contain, \overbrace{call, \ldots, call}^{C times})\}$

The above classifier for Circular dependency *is* D_T*-bounded.*

6.4.8 Alternative design anti-pattern detection methods

The average precision and recall of the proposed method is 0.87 and 0.85 respectively. Detailed experimental result is described in Appendix A.5.1. The results in specific context are comparable to current state-of-the-art solutions. When looking at individual type of a design anti-pattern, the model described in this research was outperformed by other methods described below: The detection algorithm described in [220] that was applied to Xerces dataset, when identifying the instances of the Blob anti-pattern has precision/recall at 0.95/0.95 respectively, while the model proposed in this thesis is noticeably less accurate with the result at 0.91/0.69 respectively. On the argouml dataset the same difference is 0.97/0.84 vs 0.9/0.76. The algorithm proposed in [220] is based on the support-vector machines classifier applied to the 60-dimensional space built from the values of sophisticated software metrics. Such an approach produces very accurate classification. However, kernel functions may be hard to understand for people not familiar with specific mathematical knowledge. Contrary to that, classifiers based on definable patterns use straightforward terms directly attached to the structure of the source code of the system. Such rules are easily comprehensible and modifiable for potential beneficiaries – the software engineers. A method described in [241], one of the current state-of-the-art detection algorithms, also uses similarly simple rules, but is based on richer model, which also includes lexical properties such as specific names of code entities. It outperforms our method in detecting instances of Blob in Xerces dataset in terms of recall (1.0 vs 0.69) but is slightly worse in terms of precision (0.87 vs 0.91).

In all other cases, detection algorithms based on definable patterns described in section 6.4 turn out to produce at least as accurate classification, as compared to existing research in the field: In [228] authors report a "strict accuracy" of 0.8. The average precision/recall of method described in [37] is 0.79/0.78 respectively. In [241] average precision/recall is 0.61/0.93 while at best it reaches 0.87/1.0. Method described in [287], in which additional,

Ref. method	Quality of detection with definable patterns:					
	Best			Average		
	precision	recall	F1	precision	recall	F1
[335]	0.1	0.47	0.16	0.1	0.4	0.16
[308]	0.78	0.31	0.44	0.18	0.24	0.21
[139]	0.86	1.0	0.92	0.2	0.33	0.25
[324]	1.0	1.0	1.0	0.2	0.33	0.25
[54]	1.0	0.96	0.98	1.0	0.89	0.94
[48]	1.0	1.0	0.98	0,77	0.62	0.67
[356]	1.0	1.0	0.98	0,93	1.0	0.96

Table 6.1: Comparison of the quality of detection of design patterns.

temporal information is also used to detect static patterns, the reported precision/recall is 0.78/0.61. A method described in [186], based on Bayesian network reaches 0.7/0.9 in terms of precision/recall. As a systematic review of available literature states (see [302]), the majority of published research articles on automatic identification of design anti-patterns does not provide a report on accuracy measures, or reference data (see [323], [205]), thus cannot be directly compared with this research. Yet, similar methods have also been applied to detect dual structures: i.e. design patterns. A report produced by [54] compares a few state-of-the-art detection algorithms, which when applied to individual software systems and patterns, produce excellent recall or both precision and recall (see. [241], [335], [139], [324]). Details of this comparison are summarized in Table 6.1.

6.5 Spatial and temporal relations

The following subsections define formal constructs that allow us to specify the concept of *spatial and temporal relations* between software pattern instances in software evolution.

6.5.1 Spatial relations of pattern instances

Overlapping and distance between patterns

Given two instances of patterns we will define two notions that enable us to define a *spatial relation* between them and answer the question how "close" they are to each other in the software structure. Let $T \subseteq Deps\_Types$ be a subset of types of inter-code-entity dependencies.

For any $SSn = (Ent, Deps, Metr)$ $SSn|_T$, is a subgraph of SSn with only these edges which are labeled with an element of T. Formally:

$$SSn|_T = (Ent, Deps \cap (Ent \times Ent \times T)), Metr) \tag{6.4}$$

Let $PI_1 = (V_1, E_1, Metr_1), PI_2 = (V_2, E_2, Metr_2)$ be two instances of patterns. The *overlapping* of PI_1 and PI_2 is given by:

$$Ov(PI_1, PI_2) = \frac{|V_1 \cap V_2|}{|V_1 \cup V_2|} \tag{6.5}$$

The *distance* with respect to T between PI_1 and PI_2 is given by:

$$d(PI_1, PI_2, T) = min_{v_1 \in V_1, v_2 \in V_2} dist(v_1, v_2, SSn|_T) \tag{6.6}$$

where $dist(a, b, G)$ is the distance between vertices a and b measured as the length of the shortest path between them in the multigraph G treated as undirected graph.[7]

Conceptually, the overlapping measures to what extent the two pattern instances share common vertices, and the distance measures the shortest path between any two vertices from both instances, such that the path can only be constructed with edges with labels from T. The graph-theoretic representation of the structure of software is taken directly from actual constructs of the source code. Therefore, conceptually, overlapping and distance measure how dependent on each other the respective software structures are.

The notion of distance is, in a specific sense, invariant to the revision: The formulas primarily rely on the sets of vertices of the two subgraphs, which typically appear in more than one revision. This means that we can evaluate the formula at any revision r as long as SSn_r contains all vertices from both

[7]This makes dist symmetric.

pattern instances. This yields the notion of *distance at revision* r:

$$d_r(PI_1, PI_2, T) = \begin{cases} +\infty & \text{if } (V_1 \cup V_2) \setminus Ent_r \neq \emptyset \\ min_{v_1 \in V_1, v_2 \in V_2} dist(v_1, v_2, SSn_r|_T) & \text{otherwise} \end{cases} \tag{6.7}$$

and *evolution-wide-distance*:

$$d^{evol}(PI_1, PI_2, T) = min_{r \in R_{evol}} d_r(PI_1, PI_2, T), \tag{6.8}$$

where *evol* is the evolution of the system corresponding to the revisions in R_{evol}.

Similarly, we can define *overlapping at revision* and *evolution-wide overlapping*:

$$Ov_r(PI_1, PI_2) = \begin{cases} 0 & \text{if } (V_1 \cup V_2) \setminus Ent_r \neq \emptyset \\ Ov(PI_1, PI_2) & \text{otherwise} \end{cases} \tag{6.9}$$

$$Ov^{evol}(PI_1, PI_2) = max_{r \in R_{evol}} Ov_r(PI_1, PI_2) \tag{6.10}$$

These constructs enable us to tell how distant from each other the two pattern instances are, in the entire software evolution, even if they never appear together in a single revision.
Let us formulate some obvious observations on the above definitions:

Fact 13. $Ov(PI_1, PI_2) > 0$ *iff for any* X $d(PI_1, PI_2, X) = 0$

Fact 14. $Ov_r(PI_1, PI_2) > 0$ *iff for any* X $d_r(PI_1, PI_2, X) = 0$

Fact 15. $Ov^{evol}(PI_1, PI_2) > 0$ *iff for any* X $d^{evol}(PI_1, PI_2, X) = 0$

Fact 16. *If the revisions in the evolution form a linear order* $(r_1 < r_2 < \ldots)$ *and* ev_i *is a sub-evolution corresponding to revisions* $(r_1, \ldots, r_i)$ *then for any two pattern instances* PI_1, PI_2*:*
$Ov^{ev_i}(PI_1, PI_2) \leqslant Ov^{ev_j}(PI_1, PI_2),$ *for any* $i < j$
$d^{ev_i}(PI_1, PI_2, X) \geqslant d^{ev_j}(PI_1, PI_2, X)$ *for any* $i < j, X$

6.5.2 Closeness and remoteness of pattern instances

In the following sections we will assume that there is a fixed $Dep\_Types \supseteq T = \{call, refer, type\}$, unless stated differently. The definitions from preceding sections enable us to define the notion of *closeness* of pattern instances in two flavors:

Definition 32 (closeness of pattern instances)**.** *For two pattern instances PI_1, PI_2 and the entire evolution evol we will say that:*

1. *PI_1 and PI_2 are* d-distance-close *iff* $d^{evol}(PI_1, PI_2, T) \geqslant d$
2. *PI_1 and PI_2 are* o-overlapping-close *iff* $Ov^{evol}(PI_1, PI_2) \geqslant o$

For greater simplicity, if the parameters d and o are clear from context we will say that the pattern instances are *distance-close* or *overlapping-close*. Moreover, if the type of closeness is clear, or the statement is valid for any type of closeness, we will just say that the pattern instances are *close*.

Remoteness between pattern instances

We will also consider concepts that are dual to closeness:

Definition 33 (remoteness of pattern instances)**.** *We will say that two pattern instances are* d-distance-remote *if they are not d-distance-close. We will say that two pattern instances are* o-overlapping-remote *if they are not o-overlapping-close.*

The following observations are straightforward consequences of facts 13 - 15 and the above definitions:

Fact 17. *If two pattern instances are overlapping-close then they are also d-distance-close for any positive d.*

Fact 18. *If two pattern instances are d-distance-close then they are also x-distance-close for any valid $x > d$.*

Fact 19. *If two pattern instances are d-distance-remote then they are also x-distance-remote for any valid $x < d$*

Fact 20. *Relations of distance-closeness and overlapping-closeness are symmetric and reflexive.*

6.5.3 Neighborhood and faraway of pattern instance

The relation of closeness between pattern instances can be naturally extended to *closeness* of software entity and to the set of software entities close to pattern instance:

Definition 34 (neighborhood of pattern instance)**.** *We will say that entity e is* close *(*d-distance-close*) to pattern instance PI_1, iff there exists another pattern instance $PI_2 = (V_2, E_2)$, such that PI_1 and PI_2 are d-distance-close and $e \in V_2$. The set of entities close to PI_1 will be called the* neighborhood of *PI_1.*

Definition 35. *Let V be subset of entities present in the evolution of the system. We will say that set V is* close *(*d-distance-close*) to pattern instance PI_1 iff V is the subset of neighborhood of PI_1*

Similarly, we can define dual concepts:

Definition 36 (faraway of pattern instance)**.** *We will say that entity e is* remote *(*d-distance-remote*) to pattern instance PI_1, iff there exists another pattern instance $PI_2 = (V_2, E_2)$, such that PI_1 and PI_2 are d-distance-remote and $e \in V_2$. The set of entities remote to PI_1 will be called the* faraway of *PI_1.*

Definition 37. *Let V be a subset of entities present in the evolution of the system. We will say that set V is* remote *(*d-distance-remote*) to pattern instance PI_1 iff V is the subset of faraway of PI_1*

Please note that the preceding definitions 34-37 make sense only for distance closeness (remoteness), as the defined notions become trivial for overlapping closeness.

Closeness test and remoteness test

Checking if two pattern instances are close at a certain revision is a straightforward algorithm: In case of overlapping closeness, it is just simple arithmetic on the power of sets which can be computed with elementary set-theoretical operations on the sets of pattern-instance nodes. In case of d-distance-closeness it is a matter of specific BFS traversal with bounded depth, which starts at entities of any pattern instance. We will use a formal construct named *closeness test*, which is a program that, given three arguments: two pattern instances and the software snapshot, returns 1 if these two pattern instances are close in the given software snapshot and 0 otherwise. We will additionally assume that the program can query the software snapshot about nodes and edges in SSn according to rules defined in Section 6.3.2. If the type of closeness is important we will use the term *distance-closeness test* or *overlapping-closeness test* and if the distance threshold (d) or overlapping thresholds (o) are important, we will use the term *d-distance-closeness test* or *o-overlapping-closeness test* respectively.

We will also consider the dual concept of *remoteness test* (*distance-remoteness test*, *overlapping-remoteness test*, *d-distance-remoteness test*, *o-overlapping-remoteness test*), which is a program that returns complement to 1 of the respective closeness test. Conceptually, one can think of these remoteness tests as if they were the negation of their respective closeness tests.

6.5.4 Pattern instance lifespan

Preceding sections contain definitions which lead to the notion of *closeness* - a kind of *spatial relation* between pattern instances. In the following paragraphs we will also formalize temporal relations between software patterns.

Let PI be an instance of a pattern P present in some SSn_r. If for any other r' $SSn_{r'}$ contains the same pattern instance (i.e. pattern with the same *identity* - see Section 6.3.1), then we say that r' belongs to the *lifespan* of PI. The set of all revisions that belong to the *lifespan* of PI will be denoted by $\mathscr{L}(PI)$. Conceptually, the lifespan is the time in which a particular pattern instance was present in the system. Clearly, each lifespan can be expressed as a set-theoretical sum of maximum intervals of revisions from R, where R is the linearly-ordered set of all revisions in the evolution of the observed software. For a given pattern P, the set of all such intervals in all distinct instances of P will be called the *occurrences of* P and denoted by $Occ(P)$ and each such interval will be called the *occurrence of* P. Clearly, $\mathscr{L}(PI) = \bigcup_{l \in Occ(PI)} l$.

6.5.5 Temporal and spatio-temporal relations

Let PI_1 and PI_2 be two different pattern instances of two different patterns P_1 and P_2. Given two intervals $l_1 \in Occ(PI_1)$ and $l_2 \in Occ(PI_2)$, we can name a *temporal relation* between them in terms of the Allen algebra operators (see Section 4.3.1, [30]). This allows us to define the concept of *spatio-temporal relations.*

Definition 38 (close-spatio-temporal relation)**.** *Let PI_1, PI_2, l_1, l_2 be defined as above and let A be Allen operator. Pairs (PI_1, l_1) and (PI_2, l_2) are in*

A-close-spatio-temporal *relation iff PI_1 and PI_2 are close and l_1 and l_2 are in A relation.*

When the intervals are not important we will say that PI_1 and PI_2 are in A-close-spatio-temporal *relation.*

Definition 39 (remote-spatio-temporal relation)**.** *Given symbols as in Definition 38, we will say pairs (PI_1, l_1) and (PI_2, l_2) are in* A-remote-spatio-temporal *relation iff PI_1 and PI_2 are remote and l_1 and l_2 are in A relation.*

When the intervals are not important we will say that PI_1 and PI_2 are in A-remote-spatio-temporal *relation.*

Fact 21 (inverted spatio-temporal-relation)**.** *For any pattern instances PI_1, PI_2 and their respective occurrences l_1, l_2*

1. *(PI_1, l_1) and (PI_2, l_2) are in A-close-spatio-temporal relation iff (PI_2, l_2) and (PI_1, l_1) are in A^{-1}-close-spatio-temporal relation*
2. *(PI_1, l_1) and (PI_2, l_2) are in A-remote-spatio-temporal relation iff (PI_2, l_2) and (PI_1, l_1) are in A^{-1}-remote-spatio-temporal relation*

If we look at a single occurrence of a pattern instance, we may analyze how other pattern instances are related to it in a spatio-temporal manner. We will formalize these relations with the notion of the *relative occurrences of a pattern*:

Definition 40 (close relative occurrences)**.** *Let $l_1^{PI_1}$ be an occurrence of pattern instance PI_1, A be Allen operator.*

$Occ^{close}_{[l_1^{PI_1}, A]}(P_2)$ denotes the set of all occurrences $(PI_2, l_2^{PI_2})$ in the evolution such that:

- *$PI_2 \neq PI_1$,*
- *The type of PI_2 is P_2,*
- *$(PI_1, l_1^{PI_1})$ and $(PI_2, l_2^{PI_2})$ are in A-close-spatio-temporal relation.*

The set $Occ^{close}_{[l_1^{PI_1}, A]}(P_2)$ will be called close occurrences of P_2 relative to $l_1^{PI_1}$ by A.

Definition 41 (remote relative occurrences)**.** *Let $l_1^{PI_1}$ be an occurrence of pattern instance PI_1, A be Allen operator.*

$Occ^{remote}_{[l_1^{PI_1}, A]}(P_2)$ denotes the set of all occurrences $(PI_2, l_2^{PI_2})$ in the evolution such that:

- *$PI_2 \neq PI_1$,*
- *The type of PI_2 is P_2,*
- *$(PI_1, l_1^{PI_1})$ and $(PI_2, l_2^{PI_2})$ are in A-remote-spatio-temporal relation.*

The set $Occ^{remote}_{[l_1^{PI_1},A]}(P_2)$ will be called remote occurrences of P_2 relative to $l_1^{PI_1}$ by A.

Example 1. *The example depicted in Figure 6.3 conceptually explains some of the above notions: Pattern Instance PI or type P is observed only at revision Rev_n, therefore its occurrences consist only of a single interval, which contains only Rev_n. Let us denote this interval l_{PI}. Consequently, $Occ(PI) = \{(Rev_n, Rev_n]\}$ and $\mathscr{L}(PI) = \{Rev_n\}$.*

*Another pattern instance PI' of type P' is present only in revision Rev_1 (let us denote its only occurrence as $l_{PI'}$). Since $Rev_1 < Rev_2 < Rev_n$, $l_{PI'}$ and l_{PI} are in the $<$ (*takes place before*) Allen's relation. The darker oval represents a 1-distance-neighborhood of PI, which, since all entities that constitute PI in Rev_n are present in all revisions in the software evolution, can be mapped to all these revisions. In Rev_1 some of entities of PI' are also part of this neighborhood. Consequently, PI and PI' are in 1-distance-closeness spatial relation. Moreover, $Occ^{close}_{[l^{PI},'<']}(P') = \{l_{PI'}\}$ and $Occ^{close}_{[l^{PI'},'>']}(P) = \{l_{PI}\}$. According to definitions 40 and 41, all other $Occ^{x}_{[y,z]}(t)$ sets, for all valid x, y, z, t are empty.*

The spatio-temporal relation relays on the concept of spatial relation (i.e. closeness or remoteness) and temporal relations (i.e. Allen operators). In the following sections we will show that both can be efficiently evaluated in an adaptive manner along software evolution.

6.5.6 Adaptive mining of spatial relations

A spatial relation is technically either a closeness or remoteness relation between pattern instances. It is a symmetric and reflexive binary relation. In this section we will prove that the information about spatial relations can be easily computed adaptively, along with software evolution. Technically, the remoteness relation is complement of the closeness relation[8]. Therefore, if we

[8]Please note that in the experiments described in this thesis the closeness relation is always computed with parameters other than the remoteness relation. We use

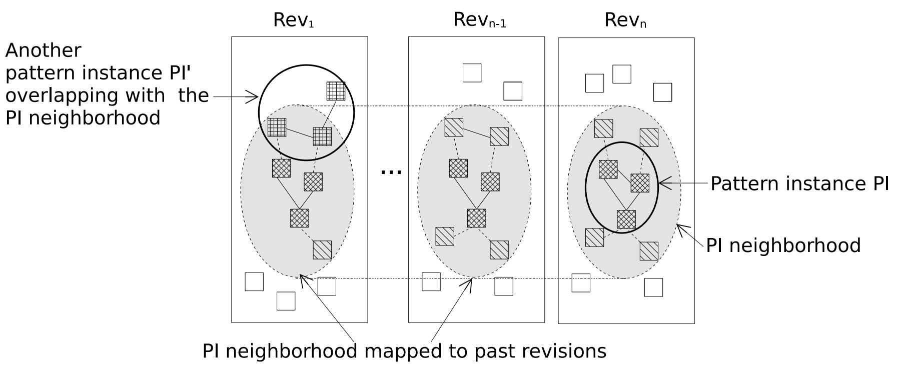
Rev₁
Rev_{n-1}
Rev_n
Another pattern instance PI' overlapping with the PI neighborhood
...
Pattern instance PI
PI neighborhood
PI neighborhood mapped to past revisions

Figure 6.3: The concept of spatio-temporal relation between pattern instances: Pattern instances PI and PI' are in 'takes place before'-close-spatio-temporal relation, with respect to 1-distance-closeness.

are able to efficiently compute the closeness relation, we are also able to compute the remoteness relation. For this reason, the arguments in the following paragraphs will focus on the closeness relation only. The remoteness relation will be mentioned only where the duality plays a non-trivial role.

Definition 42 (d-neighborhood of pattern instance in sub-evolution)**.** *Let* $\mathcal{N}_i^d(PI)$ *denote the set of entities that were at most d-distance-close to some entity from* PI *in the sub-evolution* ev_i*, corresponding to revisions* $(r_1, \ldots, r_i)$*. Formally:* $e \in \mathcal{N}_i^d(PI)$ *iff there exists a revision* r_j*, such that* $j \leqslant i$ *and at revision j there is entity* e_{PI}*, which is a node of* PI *and is d-distance-close to e. We will call* $\mathcal{N}_i^d(PI)$ *the* d-neighborhood of PI in sub-evolution $(r_1, \ldots, r_i)$[9].

The following facts are straightforward consequences of Definitions 42 and 32:

Fact 22. *If any two pattern instances are d-distance-close then their d-neighborhoods have non-empty intersection in some sub-evolution.*

Fact 23. *If two pattern instances are overlapping-close then their 0-neighborhoods have non-empty intersection in some sub-evolution.*

d_c*-distance-closeness* and d_r*-distance-remoteness* relation where $d_c \leqslant d_r - 1$. Therefore, in each experiment which is based on both closeness (C) and remoteness (R), the R is actually dual to a closeness relation different than C.

[9]Please note similar Definition 34, which defines a spatial neighborhood at single revision only. Here we define it for subevolution.

Fact 24. *For any fixed PI , $(\mathscr{N}_i^d(PI), \mathscr{N}_{i+1}^d(PI), \ldots)$ form a set-theoretical chain, i.e. $\mathscr{N}_i^d(PI) \subseteq \mathscr{N}_{i+1}^d(PI)$ for any valid revisions r_i, r_{i+1}*

Fact 25. *Any fixed pattern instances PI_1, PI_2 are d-distance-remote in sub-evolution $(r_1, \ldots, r_i)$ if none of the nodes of PI_2 is the element of $\mathscr{N}_i^d(PI_1)$.*

Adaptive evaluation of pattern instance neighborhoods

We will show that $\mathscr{N}_{i+1}^d(PI)$ can be efficiently computed from $\mathscr{N}_i^d(PI)$ and some additional information available only in $SSn_{r_{i+1}}$:

If there is a pattern instance PI that is present in revision r_{i+1} and was not present in any previous revisions, then $\mathscr{N}_{i+1}^d(PI)$ is simply the set of entities d-distance-close to entities of PI in $SSn_{r_{i+1}}$.

If PI was present in r_i and none of its entities is d-distance close to an entity affected by revision r_{i+1}, then $\mathscr{N}_{i+1}^d(PI)$ must be equal to $\mathscr{N}_i^d(PI)$.

If PI was present in r_i and at least one of its entities is d-distance-close to an entity affected by revision r_{i+1}, then $\mathscr{N}_{i+1}^d(PI)$ can be computed by finding nodes of $SSn_{r_{i+1}}$ which are connected by a path not longer than d to some entity of PI at r_{i+1}.

The consequence of the preceding statements is that, given that $\mathscr{N}_i^d(PI)$ is known, in order to find $\mathscr{N}_{i+1}^d(PI)$, it is sufficient to traverse a subgraph at revision r_{i+1}, with entities that are at most d-distance close to entities affected by revision r_{i+1} or are d-distance close to PI. This is typically a very small graph. This statement is empirically validated in Experiment A.4.1.

The above considerations can be concluded with the statement that d-neighborhood of any pattern instance can be efficiently and adaptively computed along the evolution of the system.

Adaptive evaluation of close pattern instances

Let $\{r_1, r_2, \ldots\}$ denote a linearly-ordered set of revisions with a convention: $r_i < r_j$ iff $i < j$, let ev_i denote the sub-evolution of the system corresponding to the revisions $r_1, \ldots r_i$, and let $\mathscr{C}_i$ denote the set of all pairs of different pattern instances that were close in the sub-evolution ev_i. We will show that $\mathscr{C}_{i+1}$ can be easily computed from $\mathscr{C}_i$, neighborhoods of pattern instances and the closeness test.

Fact 26. *$(\mathscr{C}_i)_{i=1,2,\ldots}$ forms a set-theoretical chain, i.e. $\mathscr{C}_i \subseteq \mathscr{C}_{i+1}$ for any valid revisions r_i and r_{i+1}.*

Proof. According to the fact 16, for any $i < j, T \subseteq Dep\_Types$ and any two pattern instances PI_1, PI_2, $Ov^{ev_i}(PI_1, PI_2) \leqslant Ov^{ev_j}(PI_1, PI_2)$ and $d^{ev_i}(PI_1, PI_2, T) \geqslant d^{ev_j}(PI_1, PI_2, T)$. Closeness of PI_1 and PI_2 can either denote that their evolution-wide overlapping is over a certain fixed threshold or that their evolution-wide distance is below a certain fixed threshold. Consequently, if $\{PI_1, PI_2\} \in \mathscr{C}_i$ then $\{PI_1, PI_2\} \in \mathscr{C}_j$ for any $j > i$. □

As a consequence of Fact 26, in order to show how $\mathscr{C}_i$ is constructed from $\mathscr{C}_{i-1}$, it is sufficient to show what new elements are added to it at revision r_i. We will use the following Lemma 4 to find them in an efficient manner.

Lemma 4. *If PI_1 and PI_2 be two pattern instances such that:*

1. *they were not d-distance-close at revision r_{i-1},*
2. *they are d-distance-close at revision r_i,*
3. *the set of entities in PI_1 does not change at revision r_i,*
4. *the set of entities in PI_2 does not change at revision r_i,*

then there must be an entity e affected by revision r_i such that in subevolution ev_i e is in d-distance-neighborhood of PI_1 and d-distance-neighborhood of PI_2.

Proof. Suppose that PI_1 and PI_2 are not d-distance-close at revision r_{i-1} and this condition does not hold at revision r_i. Then at revision r_i there is a non-directed path that starts at some entity of PI_1, has at most d edges and ends at some entity of PI_2. If at revision r_i there is no affected entity that is in the d-distance-neighborhood of PI_1, then no path of length d starting at entity from PI_1 contains an affected entity. Consequently, all such paths must contain exactly the same entities as in revision r_{i-1}. Therefore, PI_2 cannot be reached from any entity of PI_1 by such path. This leads to a contradiction with the assumption that PI_1 and PI_2 are close. Therefore, there must be an e affected by revision r_i such that e is in d-distance-neighborhood of PI_1. Since the definition of closeness is symmetric (see Fact 20), the same entity must also be in d-distance-neighborhood of PI_2. □

If a pair $(PI_1, PI_2) \in \mathscr{C}_i \setminus \mathscr{C}_{i-1}$ then pattern entities PI_1 and PI_2 were not close in r_{i-1} but are close in r_i. According to Lemma 4, the set of entities in one of the pattern instances changed or there is an entity e that is affected by revision r_i and is in d-distance-neighborhood of both instances. This also applies to overlapping closeness, according to Facts 13-15 and 18.

The above considerations allow us to outline a pseudo-algorithm that produces $\mathscr{C}_i$ from $\mathscr{C}_{i-1}$:

First, the pseudo-algorithm finds pattern instances that might be close to a pattern instance at revision r_i but were not close to it in revision r_{i-1}. We will call this set *candidate instances*. Clearly, it contains:

- pattern instances that were not present in r_{i-1} but are present in r_i
- pattern instances whose set of nodes changed in revision r_i
- pattern instances that are d-distance-close to an entity affected by revision r_i

According to Lemma 4, candidates do not contain any other elements.

Next, the algorithm finds d-distance-neighborhood of each element from candidate instances and checks if it contains at least one entity from another pattern instance. If it does, and the corresponding pair of pattern instances is not an element of $\mathscr{C}_{i-1}$ then it is added to the set of *candidate pairs*. According to Facts 13-15, 18 and 22 and 24, $\mathscr{C}_i \setminus \mathscr{C}_{i-1}$ cannot contain any other elements.

Finally, each element of candidate pairs is tested against closeness test. If such pair passes the test, it is added to $\mathscr{C}_i$.

This pseudo-algorithm, in order to produce $\mathscr{C}_i$, needs to know: 1. $\mathscr{C}_{i-1}$ 2. Affected entities 3. Their d-distance-neighborhoods 4. d-distance-neighborhoods of pattern instances that were not present in revision r_{i-1} or whose set of nodes changes in revision r_i.

Experiments A.2.2 and A.4.1 show that the first two sets are typically very small, Experiment A.4.2 and A.3 show that the last two sets are also typically very small. This makes the proposed adaptive method efficient in practical application.

6.5.7 Adaptive mining of temporal relations

The preceding Subsection 6.5.6 explained how spatial relations can be efficiently computed in an adaptive manner along the evolution of software. In this section we will show that also temporal relations can be computed likewise.

Adaptive evaluation of pattern instances lifespans

As we observe a certain pattern instance PI in the evolution at a point in time r_i we know if it was present in revisions $r_1, \dots r_i$, but we do not yet know if it will be present at r_{i+1}. Therefore, we do not know if an interval that ends at r_i is a part of the occurrences of PI. This can be decided only at revision r_{i+1}, given that PI is not present in it. This observation allows us to construct a sweep line method for adaptive evaluation of pattern instances lifespans. To simplify notions described in this and the next sections we will assume that the evolution of the analyzed system starts with artificial $r_{initial}$ revision which precedes all other revisions and ends with artificial r_{final} that follows all other revisions, such that $SSn_{r_{initial}}$ and $SSn_{r_{final}}$ are empty and consequently do not contain any pattern instances.

Let the triplet $O = (PI, r_{start}, r_{end})$ denote the fact that pattern instance PI occurred in all revisions $[r_{start}, r_{end}]$ and the interval is maximal, i.e. both conditions hold:

1. PI was not present in $r_{start-1}$

2. PI was not present in r_{end+1}

Let $\mathscr{O}'_i$, denote the set of all such facts that were true in the sub-evolution from revision r_1 to revision r_i. By convention we will consider $\mathscr{O}'_0 = \emptyset$

Let $\mathscr{O}_i$, denote the set of all triplets (PI, r_{start}, r_{end}) such that there is a pattern instance PI and the interval (r_{start}, r_{end}) is the element of $Occ(PI)$[10] in the sub-evolution from revision r_1 to revision r_{i-1}. By convention we will consider $\mathscr{O}_0 = \mathscr{O}_1 = \emptyset$.

There is a subtle, yet important difference between $\mathscr{O}'_i$ and $\mathscr{O}_i$: As $\mathscr{O}_i$ refers to the evolution that ends at r_{i-1} and it encodes actual occurrences of pattern instances in this evolution, it can be properly evaluated at revision r_i, as we know which pattern instances "disappeared" at this revision. Contrary to that, $\mathscr{O}'_i$ refers to the evolution that ends at r_i and it also encodes the candidates for occurrences of pattern instances that may become actual occurrences in later revisions.

We will show how the pair $(\mathscr{O}_{i+1}, \mathscr{O}_{i+1})$ can be adaptively evaluated from $\mathscr{O}'_i$ and $\mathscr{O}_i$.

[10]see Section 6.5.4 for the definition of the *lifespan* and occurrences (Occ) of pattern instance.

Fact 27. *$\mathscr{O}_i \subseteq \mathscr{O}'_i$, for any valid revision r_i*

Fact 28. *$(\mathscr{O}_i)_{i=1,2,\ldots}$ forms a set-theoretical chain, i.e. $\mathscr{O}_i \subseteq \mathscr{O}_j$ for any $i < j$ and valid revisions r_i, r_j.*

Theorem 5. *The pair $(\mathscr{O}_{i+1}, \mathscr{O}'_{i+1})$ can be computed from the pair $(\mathscr{O}_i, \mathscr{O}'_i)$ and the set of pattern instances at revision r_{i+1}.*

Proof. For any pattern instance PI, the following statements hold:

- If PI is present at r_{i+1} and it was not present at r_i then $(PI, r_{i+1}, r_{i+1}) \in \mathscr{O}'_{i+1} \setminus \mathscr{O}'_i$
- If PI is present at r_{i+1} and it was present at r_i then a $\mathscr{O}'_i$ contains exactly one element (PI, x, r_i) for some x. This element is not member of $\mathscr{O}'_{i+1}$, but (PI, x, r_{i+1}) is.
- If PI is not present at r_{i+1} and it was present at r_i then a $\mathscr{O}'_i$ contains exactly one element (PI, x, r_i) for some x. In this case $(PI, x, r_i) \in \mathscr{O}_{i+1} \setminus \mathscr{O}_i$ and $(PI, x, r_i) \in \mathscr{O}'_{i+1}$.

Please note that at revision r_{i+1} for any pattern instance PI we can check if it was present at r_i, since this is equivalent to the existence of element (PI, x, r_i) in $\mathscr{O}'_i$ for some x. Therefore, if we know pattern instances present at revision r_{i+1}, we can easily transform pair $(\mathscr{O}_i, \mathscr{O}'_i)$ to $(\mathscr{O}_{i+1}, \mathscr{O}'_{i+1})$, according to the rules derived from the above three statements:

- If PI is present at r_{i+1} and it was not present at r_i then a new triplet (PI, r_{i+1}, r_{i+1}) must be added to $\mathscr{O}'_i$.
- If PI is present at r_{i+1} and it was present at r_i then a $\mathscr{O}'_i$ contains exactly one element (PI, x, r_i) for some x. This element must be replaced with (PI, x, r_{i+1}) in $\mathscr{O}'_{i+1}$.
- If PI is not present at r_{i+1} and it was present at r_i then a $\mathscr{O}'_i$ contains exactly one element (PI, x, r_i) for some x. In this case (PI, x, r_i) must be added to $\mathscr{O}_{i+1}$.

According to the definitions of $\mathscr{O}'_i$ and $\mathscr{O}_i$ and Facts 27-28, applying these three rules on pairs for all pattern instances present at r_i or r_{i+1} on $(\mathscr{O}_i, \mathscr{O}'_i)$ transforms it to $(\mathscr{O}_{i+1}, \mathscr{O}'_{i+1})$ $\square$

Theorem 5 yields a simple adaptive algorithm for the evaluation of *occurrences* for all pattern instances in the software evolution. Indeed: it is sufficient to start with empty sets $\mathscr{O}'$ and $\mathscr{O}$, and transform them with each revision according to the rules described in the proof. It is necessary to know pattern instances at each revision, but this can also be done in an adaptive manner, according to Theorem 3 from Section 6.3.5.

6.6 Mining spatio-temporal patterns in software

6.6.1 Spatio-temporal rules

In Sections 6.5.1 and 6.5.5 we have defined the concept of a spatio-temporal relation between pattern instances. In this section we define the related notion of *spatio-temporal rule.*

Definition 43 ((A, l, d)-neighborhood)**.** *Let PI be some pattern instance of pattern P and let l be some maximal interval from the lifespan of PI, A be some Allen algebra operator and $d \geqslant 0$ be the distance threshold. Let $\mathscr{E}nt$ denote the set of all entities that were present in at least one revision in the evolution of the system and let R denote the set of all revisions in this evolution. The* (A, l, d)-neighborhood *of PI is the set of pairs $(E, (r_1, r_2))$, $E \in P(\mathscr{E}nt), r_1, r_2 \in R$ such that:*

1. *for any entity $e \in E$, e is present in all revisions from r_1 to r_2,*
2. *l and (r_1, r_2) are in A relation,*
3. *E is d-distance-close to PI.*

Definition 44 ((A, l, d)-faraway of pattern instance)**.** *Let PI, A, l, d and $\mathscr{E}nt$ be given as in Definition 43. The* (A, l, d)-faraway *of PI is the set of pairs $(E, (r_1, r_2))$, $E \in P(\mathscr{E}nt), r_1, r_2 \in R$ such that:*

1. *for any entity $e \in E$, e is present in all revisions from r_1 to r_2,*
2. *l and (r_1, r_2) are in A relation,*
3. *E is d-distance-remote to PI.*

If the distance threshold d is clear from context, we will use simpler notation: (A, l)-neighborhood and (A, l)-faraway.

Fact 29. *Let A, P, l, PI be defined as above. (A, l, d_n)-neighborhood of PI and (A, l, d_f)-faraway of PI are disjoint for any distance thresholds such that $d_f \geqslant d_n \geqslant 0$.*

The notions of (A, l, d)-neighborhood (A, l, d)-faraway may appear very complex, but conceptually they just define "spatio-temporal areas" in the entire software evolution where one could find an occurrence of other pattern instance which is in respectively: A-close- and A-remote-spatio-temporal relation. This intuitive concept is formalized in the following facts:

Fact 30. *Let PI, A, l and d be defined as in Definition 43 and let l_{PI_2} be an occurrence of some pattern instance PI_2 different than PI. The following statements are equivalent:*

- *(PI_2, l_{PI_2}) is in A-d-distance-closeness spatio-temporal relation with (PI, l),*
- *there is pair $(E, (r_1, r_2))$ in (A, l, d)-neighborhood of PI, such that PI_2 is 0-overlapping-close to E and $l_{PI_2} = (r_1, r_2)$.*

Fact 31. *Let PI, A, l and d be defined as in Definition 43 and let l_{PI_2} be an occurrence of some pattern instance PI_2 different than PI. The following statements are equivalent:*

- *(PI_2, l_{PI_2}) is in A-d-distance-remoteness spatio-temporal relation with (PI, l),*
- *there is pair $(E, (r_1, r_2))$ in (A, l, d)-faraway of PI, such that PI_2 is 0-overlapping-close to E and $l_{PI_2} = (r_1, r_2)$.*

If we take one type of software pattern (e.g. YOYO), and then consider all its instances in the evolution of the system, we may compute respective $(A, l_?, d)$-neighborhoods and $(A, l_?, d)$-faraways for these instances, for some fixed A and d. Clearly, these spatio-temporal areas will "cover" a certain fragment of the entire evolution.

The question arises, how big this fragment is. If it is very small, then we may hypothesize that typically instances of this type of pattern (YOYO) are much more likely to appear only in specific areas of the source code.

This conceptual description is a motivation for the concept of spatio-temporal rule, which is formalized in the following paragraphs in two manners: quantitative and qualitative. In the first, we just consider existence of at least one (A, l, d)-neighborhood that covers a given area. In the more advanced second one, we also take into consideration how many such different (A, l, d)-neighborhoods cover this area.

For the sake of simplicity in the following paragraphs we assume that there are two fixed distance thresholds: $d_r > d_c > 0$ (respectively: remoteness and closeness) and we simplify some of the notions: Unless stated explicitly: d_c-*distance-closeness* will be called just closeness, d_r-*distance-remoteness* will be called remoteness, (A, l, d_c)-*neighborhood* will be called (A, l)-neighborhood, (A, l, d_r)-$faraway$ will be called (A, l)-faraway, etc.

Qualitative approach

Definition 45. *Let T be a type of software pattern, A be Allen-algebra operator, Ev be the software evolution, $s \in \{closeness, remoteness\}$. A triplet (T, A, s) will be called a spatio-temporal* clause. *An* interpretation *of the clause in Ev, denoted as $\models_{Ev} (T, A, s)$ is:*

- *the sum of all (A, l)-neighborhoods of pattern instances PI such that: PI is of type T and l is the maximum interval in the lifespan of PI in Ev, if $s = closeness$ or*
- *the sum of all (A, l)-$faraways$ of pattern instances PI such that: PI is of type T and l is the maximum interval in the lifespan of PI in Ev, if $s = remoteness$*

A conjunction of clauses $(T_1, A_1, s_1), \ldots, (T_n, A_n, s_n)$, denoted by $(T_1, A_1, s_1) \wedge \ldots \wedge (T_n, A_n, s_n)$ will be interpreted in the Ev as $\bigcap_{i=1,\ldots,n} \models_{Ev} (T_i, A_i, s_i)$.

Definition 46 (spatio-temporal rule)**.** *An expression*

$$(T_1, A_1, s_1) \wedge (T_2, A_2, s_2) \wedge \ldots \wedge (T_n, A_n, s_n) \rightarrow T_d,$$

where T_d is a type of software pattern will be called spatio-temporal rule. *Let $PI = (V_{PI}, E_{PI})$ be a pattern instance of type T, and l_{PI} be a maximal interval which defines one occurrence of this pattern. We will say that this* occurrence is covered by this spatio-temporal *rule iff there exists a pair (E, l)*

in the interpretation of the left-hand side of the rule such that PI is 0-overlapping-close *to E and $l_{PI} = l$.*

If, additionally $T = T_d$, we will say that this pattern instance supports *this spatio-temporal rule.*

The notions of clause, rule and their interpretation are explained conceptually in Example 2 below.

Quantitative approach

In the previous paragraph we have constructed a spatio-temporal rule with the use of a clause that was interpreted as existence of at least one spatio-temporal relation. A question arises how many such relations actually exist. This leads to the concept of a *quantitative clause*:

Definition 47. *Let T, A, Ev, s be defined as in Definition 45. Moreover, let $N \subseteq \mathbb{N}$ be a sub-set of natural numbers.*

A triplet (T, A, s, N) will be called a quantitative clause. Its interpretation in Ev (denoted as $\models_{Ev} (T, A, s, N)$) will be given by the following definition: $(E, l) \in (\models_{Ev} (T, A, s, N))$ iff there exists $n \in N$ and n different pattern instances of type T $(PI_1, \ldots, PI_n)$ with respective maximal lifespan intervals $(l_1, \ldots, l_n)$ such that:

- *(E, l) is in (A, l_i)-neighborhood of PI_i for each $1 \leqslant i \leqslant n$, if $s =$ closeness or*
- *(E, l) is in (A, l_i)-$faraway$ of PI_i for each $1 \leqslant i \leqslant n$, if $s =$ remoteness.*

Fact 32. $(\models_{Ev} (T, A, s, \{1\})) = (\models_{Ev} (T, A, s))$

Informally, we can say that $(\models_{Ev} (T, A, s, \{n\}))$ is a space of spatio-temporal areas in the software evolution that are in respective spatio-temporal relation to n different pattern instances of type T.

The conjunction of clauses and spatio-temporal rule can be defined in the quantitative approach analogously: For a given rule

$$(T_1, A_1, s_1, N_1) \wedge \ldots \wedge (T_n, A_n, s_n, N_n) \rightarrow T_d$$

and a pattern instance $PI_d = (V_{PI_d}, E_{PI_d})$ of type T present in this evolution during some maximal interval l such that there exists a pair (E, l) in the interpretation of the left-hand side of the rule and PI_d is $0{-}overlapping{-}close$

to E, we will say that this occurrence is *covered* by this rule. Additionally, when $T = T_d$ we will say that this occurrence *supports* this rule.

The notions of clause, rule and their interpretation are explained conceptually in Example 2 below.

The previous paragraphs define the notion of a spatio-temporal rule and its interpretation in the system evolution. These construct allow us to use spatio-temporal rules to predict occurrences of design anti-patterns in the system evolution. In the following sections we explain how the rules can be mined in the course of the software development process.

6.6.2 Construction of decision table

In this section we will describe a method for mining spatio-temporal rules from the data that encodes all spatio-temporal relations between pattern instances in the evolution of the system. The following paragraphs describe the steps that lead to a set of such rules.

At first, the knowledge about spatio-temporal relations is formalized as a decision table. Conceptually, each row in this table corresponds to a single continuous interval from the lifespan of a pattern instance PI of some pattern P. The type of P stands for the decision in this row and other attributes describe the number of spatio-temporal relations of PI and other pattern instances in the evolution of the analyzed system. Formally, other attributes (i.e. condition attributes) are labeled by the pair $(A, t)^{close}$ or $(A, t)^{remote}$ where A is one of the non-inverse Allen's algebra relations (see Section 4.3.1) and t is the type of software pattern. In the row that represents an interval $l_1^{PI_1}$ of a pattern, the value of attribute $(A, t)^{close}$ is the power of $Occ^{close}_{[l_1^{PI_1}, A]}(t)$ set and the value of attribute $(A, t)^{remote}$ is the power of $Occ^{remote}_{[l_1^{PI_1}, A]}(t)$ set[11]. We will call these types of conditional attributes the *closeness attributes* and the *remoteness attributes* respectively. Definition of $Occ^{close}_{[l_1^{PI_1}, A]}(t)$ and $Occ^{remote}_{[l_1^{PI_1}, A]}(t)$ relies on the underlying closeness relation: the former directly, while the latter indirectly, by a dual remoteness relation. Please note that these are two different relations: technically they are d_c-distance-closeness and d_r-distance-remoteness for some fixed $d_c \leqslant d_r - 1$, as mentioned earlier in this chapter.

Conceptually, the greater the power of the former set, the more related the

[11]These two sets are defined in Section 6.5.4 in Definitions 40 and 41.

occurrence represented by the row to other instances by the corresponding close spatio-temporal relation. Similarly, the greater the power of the latter set, the more related the occurrence represented by the row to other instances by the corresponding remote spatio-temporal relation.

Please note that the consequence of Fact 32 is that this decision table encodes the information about spatio-temporal relations in both a quantitative and a qualitative manner, as the concepts defined in Section 6.6.1. Indeed: the value in each cell encodes the number of respective spatio-temporal relations which yields a quantitative view. Yet a discretization of the table in which each value v is changed to 1 iff $v > 0$ produces a corresponding decision table with a binary attribute which yields a qualitative view.

Example 2. *Suppose that in the evolution of the system with revisions* $(r_1, \ldots, r_6)$ *only the following pattern instances were observed:*

1. *Instance of Blob pattern observed in revisions* r_5 *and* r_6*, denoted by* $Blob_1$*,*

2. *instance of YoYo pattern observed in revisions* r_1 *and* r_2*, denoted by* $YoYo_1$*,*

3. *instance of YoYo pattern observed in revisions* r_2 *and* r_3*, denoted by* $YoYo_2$*.*

4. *instance of Swiss Army Knife (SAK) pattern observed in revisions* r_1 *-* r_3*, denoted by* SAK_1*.*

Moreover, let us suppose that $Blob_1$ *is close to* $YoYo_1$ *and close to* $YoYo_2$ *and it is remote to* SAK_1 *in the evolution. In such a setting, the row corresponding to the pattern instance* $Blob_1$ *has the form presented in Table 6.2:*

(Meets, YoYo)$^{\text{cl.}}$	(Before, YoYo)$^{\text{cl.}}$	(Before, SAK)$^{\text{rem.}}$	...	**decision**
0	2	1	0 ... 0	'Blob'

Table 6.2: Exemplary row from the decision table that encodes all spatio-temporal relations in the software evolution. SAK denotes Swiss Army Knife, cl. denotes close and rm. denotes remote.

Since the only two closeness-spatio-temporal relations observed in the evolution both correspond to two distinct facts that an instance of <u>YoYo</u> pattern is before *$Blob_1$, the value in the corresponding cell is 2. Similarly, the evolution contained only one remote-spatio-temporal relation: $Blob_1$ is* remote *to instance of* Swiss Army Knife *SAK_1, which was observed* before *$Blob_1$, the value of the corresponding cell is 1. As no other instances are in any spatio-temporal relation with $Blob_1$, all remaining cells have value 0.*

Let us consider the following quantitative and quantitative spatio-temporal rules:

$$(YoYo, Before, cl.) \wedge (SAK, Before, rem.) \rightarrow Blob \quad (6.11)$$

$$(YoYo, Meets, cl.) \wedge (YoYo, Before, cl.) \wedge (SAK, Before, rem.) \rightarrow YoYo \quad (6.12)$$

$$(YoYo, Before, cl., \{2\}) \rightarrow SAK \quad (6.13)$$

$$(YoYo, Before, cl., \{3, 4, 5\}) \rightarrow SAK \quad (6.14)$$

- *Qualitative rule 6.11 covers $Blob_1$ and is supported by it, which is equivalent to the fact that the occurrence of $Blob_1$ is in the interpretation of this rule (see Definition 46).*

- *Qualitative rule 6.12 has empty interpretation, since all elements in the interpretation of its first clause has form $(X, (r_3, y))$ or $(X, (r_4, y))$ for some X and y and all elements in the interpretation of the last clause has form $(X, (r_5, y))$ or $(X, (r_6, y))$ for some X, y. Thus, their intersection is empty.*

- *Quantitative rule 6.13 is covered by $Blob_1$ but it does not support it (see Definition 47). The interpretation of this rule consist of valid pairs $(E, (r_{start}, r_{end}))$, where $r_{start} \in \{r_5, r_6\}$ and E is a set of valid entities which are elements of $\mathcal{N}_6(YoYo_1) \cap \mathcal{N}_6(YoYo_2)$ (see Definition 42).*

- *The interpretation of the left hand side of the quantitative rule 6.14 is empty, as the evolution contains only two distinct occurrences of YoYo instances.*

6.6.3 Decision table with arbitrary rows

In the above construction of the decision table both rows and columns may be said to correspond to the static patterns of a specific type (practically to design anti-patterns): Each row corresponds to a single occurrence of pattern instance and its type defines the decision, whereas each column is related to specific spatio-temporal relations with static patterns of specific type. One can imagine a different definition of the table, where the rows represent the occurrences of static patterns of a different type to those which define the columns. In fact, in order to construct such a table, we need to be able to specify sets $Occ^{close}_{[l_1^{PI_1},A]}(t)$ and $Occ^{remote}_{[l_1^{PI_1},A]}(t)$ for each row. Both these sets rely on the following concepts: type and unique identity (Section 6.3.1), closeness (Section 6.5.2), lifespan (Section 6.5.4). In other words, if we define these concepts for a set of subgraphs of $(SSn)_i$, we can represent them as rows in the decision table, according to the above definitions. In the following paragraph, we will do so in order to introduce the concept of *random equivalent*. We will also use a similar approach in other applications (see Section 6.7.4).

Random equivalents

The decision table described in Section 6.6.2 is used to mine spatio-temporal rules. Supervised machine learning algorithms require labeled training data where each example is given an appropriate decision class (label). In general, the number of examples with different decision classes should be similar so that the mining algorithm does not produce a biased model. In fact, the problem of unbalanced training data is a separate, broad area of knowledge (see [170]).

In some experiments described later in this section, we try to train a model that will discern occurrences of certain design anti-patterns from occurrences of other similar subgraphs of SSn. In order to do so, we will introduce a special kind of a static pattern, called *random equivalent*.

Definition 48 (Random equivalent)**.** *For a given pattern instance PI contained in software snapshot SSn, such that the type of PI is t, the* random equivalent *of PI is every containment-complete subgraph G of SSn such that: 1. The number of nodes in G is equal to the number of nodes in PI 2. G is not an element of the upper bound of t in SSn.*

The conventions described in the following paragraphs will allow us to introduce the notions of: *type*, *uniqueness*, *occurrence* and *lifespan* of random equivalent, so that they can be treated uniformly with other static patterns in the algorithms described in this chapter:

Let G be a random equivalent of a pattern instance PI of type t. We will consider that the *type* of G is NOT_t (e.g. NOT_Blob, when PI is an instance of Blob). Please recall from Section 6.3.1 that each pattern instance is uniquely identified by its type and the name of its main entity. We will *uniquely identify* each random equivalent by its type and the set of names of entities that comprise its vertices. We will say that such random equivalent *is present* in revision r iff all its nodes are present in SSn_r and a containment-complete subgraph induced by these nodes is not in the upper bound of t in SSn_r. This definition of the presence of random equivalent at a given revision is the basis of the notions of *occurrence* and *lifespan* which are defined as in Section 6.5.4. Since random equivalents at a given revision r are subgraphs of SSn_r, and we have defined their uniqueness, all variants of remoteness and closeness with corresponding remoteness and closeness tests (defined in Section 6.5.2) apply to them identically as they apply to other static patterns. Therefore, in the following paragraphs, unless explicitly stated otherwise, the term *pattern instance* may refer to both random equivalent and the instance of a design anti-pattern.

Decision table with random equivalents

As explained earlier in this section, the DT decision table encodes spatio-temporal relations of each occurrence of a static pattern with other occurrences of static patterns. As we have defined the identity, the occurrences and lifespan of random equivalent, we can relate the notion of relative occurrence of the random equivalent in the way it was defined for other static patterns in this section. Consequently, we can represent all spatio-temporal relations of any random equivalent in a separate row of the decision table described earlier in this section. However, the types of random equivalents are not used to construct additional columns.

Clearly, a typical software snapshot contains significantly more random equivalents than instances of software design anti-patterns. The motivation for adding extra rows to the decision table is to have balanced decision classes. Therefore, we will only add rows that correspond to just a few randomly selected random equivalents that resemble - to some extent - design anti-patterns. Details on the criteria of choice are described in Section 6.6.4.

A decision table with rows related to occurrences of random equivalents will be called *decision table with random equivalents*. Please note that such a decision table only contains additional rows, but the number of columns does not change, as we do not add additional columns for each type of a random equivalent.

Grouping decision classes

In the above construction of the decision table with random equivalents, each row corresponds to an individual occurrence of either random equivalent or an anti-pattern. Each of them has its type. In some situations we need to equate some types of such patterns, which corresponds to equating respective decision classes in the decision table. Arguably the simplest example is a situation when all types of anti-patterns are equated, as are all types of random equivalents. In such case we might say that the table implicitly encodes spatio-temporal areas where an instance of an anti-pattern may occur from the spatio-temporal areas where they are absent.

We assume that when two types of anti-patterns T_1 and T_2 are equated, so are the types of corresponding random equivalents: NOT_T_1 and NOT_T_2 respectively. According to the description in Section 6.6.4 below, the procedure of constructing the decision table is twofold: first, the table contains rows which correspond to actual occurrences of anti-patterns, and then it is completed with rows corresponding to randomly selected random equivalents, which ensures a balanced distribution of certain decision classes. When certain types of anti-patterns are to be equated, this second step must be carried out carefully, so that selected random equivalents are actual counter-examples of either of the equated types of anti-patterns. Therefore, in this case we require that each random equivalent of type NOT_T_1 is also outside of the upper bound of T_2.

6.6.4 Adaptive evaluation of spatio-temporal relations

In Sections 6.5.6 and 6.5.7 we have shown that pattern instances and the maximal intervals in their lifespan can be computed adaptively, along the system evolution. In this section we present a similar construct for building a decision table with random equivalents.

Let $(r_1, r_2, \ldots)$ be linearly-ordered revisions in the software evolution, where $r_i < r_j$ iff $i < j$. Let DT_i be the decision table constructed according to the method described in Section 6.6.2, which encodes all spatio-temporal relations between pattern instances that were present in sub-evolution during revisions $r_1, \ldots r_{i-1}$. By convention, we assume that D_0 is empty and we assume that DT_i contains rows that corresponds to occurrences that end not later than at revision r_{i-1}.

Fact 33. *For a fixed row and a fixed closeness conditional attribute a_c, revi-*

sions $r_i < r_j$ the value of a_c in DT_i is not greater than its value in DT_j.

Fact 34. *For a fixed row and a fixed remoteness conditional attribute a_r, revisions $r_i < r_j$ the value of a_r in DT_i is not lower than its value in DT_j.*

Let DT'_i denote the extended version of the decision table DT_i in which each row is extended with three additional attributes, which uniquely identify the occurrence of the corresponding pattern instance.
Namely: (PI, r_{start}, r_{end}), where r_{start} denotes the first revision of the occurrence, r_{end} – its last revision and PI – the pattern instance. Please note that in this context the uniqueness of the pattern instance PI is given by its identity, even if the instance can change over time (see Section 6.3.1).

In the following paragraphs we show that DT'_i can be adaptively computed from DT'_{i-1}, the set of pattern instances present at revision r_i and other structures, which can also be computed adaptively according to the descriptions in the preceding sections.

Theorem 6. *Let r_i, DT'_i, be defined as above, ev_i denote the sub-evolution of the system in revisions $(r_1, \ldots, r_i)$, $\mathscr{O}_i$ and $\mathscr{O}'_i$ be defined as in Section 6.5.7. Let $\mathscr{C}_i$ denote the set of all pairs of pattern instances that were close in ev_i, with respect to the closeness relation that defines the closeness attributes in DT'_i. Let $\mathscr{C}_i^{remote}$ denote the set of all pairs of pattern instances that were close in ev_i, with respect to the closeness relation that defines the remoteness attributes in DT'_i. DT'_i can be computed from DT'_{i-1}, $\mathscr{O}_{i-1}$, $\mathscr{C}_{i-1}$, $\mathscr{C}_{i-1}^{remote}$, $\mathscr{O}_i$, $\mathscr{C}_i$, $\mathscr{C}_i^{remote}$.*

Proof. We will show what changes need to be done to DT'_{i-1} to transform it into DT'_i.

A new row can appear in DT'_i if there is a new occurrence of a pattern instance that ends at revision r_{i-1}. It is equivalent to the fact that $\mathscr{O}_i$ contains a triplet (PI, r_{start}, r_{i-1}) that was not the element of $\mathscr{O}_{i-1}$, for the pattern instance PI and some revision r_{start}.

The three additional columns of DT'_i can be taken directly from the triplet (PI, r_{start}, r_{i-1}). A decision attribute comes directly from the type of PI. The value of a $(A, t)^{close}$ attribute is the number of other occurrences of pattern instance of type t that are in A-closeness-spatio-temporal relation with $(PI, (r_{start}, r_{i-1}))$. Each such occurrence must be represented by exactly one row of DT'_i. If we take any other row of DT'_i, where additional attributes have values (PI_2, r_1, r_2), we can tell 1. if PI and PI_2 are close in ev_i, by checking if $\mathscr{C}_i$ contains a corresponding pair. 2. what is Allen's relation between (r_{start}, r_{i-1}) and (r_1, r_2). 3. what is the type of PI_2. This information is sufficient to compute the value of each attribute $(A,t)^{close}$ in the newly added row. By analogy, we can tell if two pattern instances are remote by checking if $\mathscr{C}_i^{remote}$ does not contain a corresponding pair. Therefore we can also compute the value of each $(A,t)^{remote}$ attribute.

So far we have proven that DT'_{i-1}, $\mathscr{O}_{i-1}$, $\mathscr{O}_i$, $\mathscr{C}_i$, $\mathscr{C}_i^{remote}$ is sufficient to construct new rows in DT'_i that were not present in DT'_{i-1}.

To complete the proof we will show that changes in the conditional attributes of other rows can be updated on the basis of available information:

Suppose there is an occurrence of pattern instance PI, with a corresponding row in DT'_{i-1} with additional attributes (PI, r_{start}, r_{end}), $r_{start} < r_{end} < r_{i-1}$ and a conditional attribute labeled by $(A,t)^{close}$ such that the value of this attribute changes in this row in DT'_i. This is possible iff:

1. there is a pattern instance PI' of type t which is close to PI in ev_{i-1} and

2. an occurrence $(r_{start'}, r_{end'})$ of PI' of type t is in A Allen relation to (r_{start}, r_{end})

and at least one of the two conditions did not hold at revision r_{i-2}. It is possible iff:

1. PI' was not close to PI at revision r_{i-2} or

2. the occurrence $(r_{start'}, r_{end'})$ was not yet known at this revision (i.e. $r'_{end} = r_{i-1}$), which is equivalent to the statement that $\mathscr{O}_i \setminus \mathscr{O}_{i-1}$ contains a triplet $(PI, r_{start'}, r_{end'})$.

In the latter case we know that the triplet $(PI', r_{start'}, r_{end'})$ corresponds to a newly created row in DT'_i, which is covered above, since according to Fact 21, each spatio-temporal relation between $(PI, (r_{start}, r_{end}))$ and $(PI', (r_{start'}, r_{end'}))$ corresponds to inverted spatio-temporal relation between $(PI', (r_{start'}, r_{end'}))$ and $(PI, (r_{start}, r_{end}))$.

In the former case $\mathscr{C}_i \setminus \mathscr{C}_{i-1}$ contains a pair $\{PI, PI'\}$. If we look at row from DT'_i such that its additional attributes are $(PI', r_{start'}, r_{end'})$ for some $r_{start'}, r_{end'}$, we can determine the Allen's relation between (r_{start}, r_{end}) and $(r_{start'}, r_{end'})$, and the type of PI'. Therefore for each such case we can find a corresponding closeness attribute and increase it by 1 in the row related to the triplet (PI, r_{start}, r_{end}).

Similarly: suppose there is an occurrence of pattern instance PI, with a corresponding row in DT'_{i-1} with additional attributes (PI, r_{start}, r_{end}), $r_{start} < r_{end} < r_{i-1}$ and some conditional attribute labeled by $(A, t)^{remote}$ such that the value of this attribute changes in this row in DT'_i. This is possible iff:

1. there is a pattern instances PI' of type t which is remote to PI in ev_{i-1} and

2. some occurrence $(r_{start'}, r_{end'})$ of PI' of type t is in A Allen relation to (r_{start}, r_{end})

and at least of the two conditions did not hold at revision r_{i-2}. It is possible iff:

1. PI' was not remote to PI at revision r_{i-2} or

2. the occurrence $(r_{start'}, r_{end'})$ was not yet known at this revision (i.e. $r'_{end} = r_{i-1}$), which is equivalent to the statement that $\mathscr{O}_i \setminus \mathscr{O}_{i-1}$ contains a triplet $(PI, r_{start'}, r_{end'})$.

In the latter case we know the triplet $(PI', r_{start'}, r_{end'})$ corresponds to a newly created row in DT'_i, which is covered above.

In the former case $\mathscr{C}^{remote}_{i-1} \setminus \mathscr{C}^{remote}_i$ contains a pair $\{PI, PI'\}$. If we look at row from DT'_i such that its additional attributes are $(PI', r_{start'}, r_{end'})$ for some $r_{start'}, r_{end'}$, we can determine the Allen's relation between (r_{start}, r_{end}) and $(r_{start'}, r_{end'})$, and the type of PI'. Therefore, we can find a corresponding remoteness attribute and decrease it by 1 in the row related to the triplet (PI, r_{start}, r_{end}). $\square$

Theorems 1, 3, 5 and 6, together with supporting Fact 26, yield an adaptive algorithm for evaluating the DT' decision table along with the evolution of the system: We start with empty DT'_0 and with each revision we modify it according to the rules described in the proof of Theorem 6. In practice, the cost of computation at each revision depends on the number of entities affected by it and changes in the sets $Inst_{r_i}$ (Section 6.3.5), $\mathscr{O}_i$ (Section 6.5.7), $\mathscr{C}_i$ and $\mathscr{C}^{remote}_i$ (Section 6.5.6). The changes of these sets in subsequent revisions tend to be very small. This statement is experimentally validated in experiments described in Appendices A.2-A.4.2.

Running a regular rule-based classification algorithm on the decision table produces a set of classification rules which are in fact spatio-temporal rules. This concept is further explored in the following section, which describes a few possible applications of this model together with their empirical validation.

6.7 Empirical validation of the proposed model

This section describes how the proposed formal framework was validated empirically in a few different applications.

6.7.1 Data used in experiments

The research described in this thesis was validated on the history of the development of a few popular open source projects, whose source code is maintained in publicly available SCM, which was Subversion ([4]) or Git ([2]), and whose issues are tracked in a publicly available issue tracker, which predominantly was Jira ([3]) or Bugzilla ([22]). Some of the methods described in this thesis require both data sources. In such case, it is assumed to be synchronized by a procedure described in section 2.2.4, so that each commit is always bound to one or more issues from the issue tracker. The data that was used covers at least three years of active development from an early stage and it includes at least 5000 commits in the main development branch (with one exception described below).

Detailed information about specific software systems is given in the following list:

- ArgoUML ([5], [11], [10]) is a simple, old-fashioned UML editor, which used to be very popular. The SCM of this software (along with Xerces2j and JHotDraw) is frequently used as the source of data in mining software repositories research. The analyzed evolution of this software spans from January 1998 to December 2011.

- Wildfly ([20], [19], [18]) is a popular Java application server. The analyzed evolution of this software spans from June 2010 to June 2013.

- Xerces2j ([21], [9], [8]) is a popular Java XML Parser. The analyzed evolution of this software spans from November 1999 to May 2008.
- JHotDraw ([7], [14]) is a Java framework for 2D graphics. The analyzed evolution of this software spans from October 2000 to November 2012. The software was actively developed in this period, but the number of corresponding revisions was as little as 670.
- Elasticsearch ([13], [12]) is a popular search engine. The analyzed evolution of this software spans from February 2010 to September 2017.
- Lucene-solr ([16], [15], [8]) is a popular search engine. Its analyzed evolution spans from September 2001 to November 2016.
- Struts1 ([17], [9], [8]) is a formerly popular java web framework. Its analyzed evolution spans from May 2000 to December 2008.

6.7.2 Prediction of potential design anti-patterns

In this section we describe the method of using spatio-temporal rules to find spatio-temporal areas where certain types of design anti-patterns are likely to appear. The method is conceptually presented in Figure 6.4: Arrow 1 corresponds to the process of training the spatio-temporal rules on the training evolution, according to the description in Section 6.6.4. Arrow 2 corresponds to the interpretation of spatio-temporal rules in the context of test evolution, according to the description in Section 6.6.1. Arrow 3 corresponds to the encoding of all spatio-temporal relations from the test evolution, according to the description in Sections 6.6.2-6.5.6. Arrow 4 corresponds to the verification of the prediction quality described below. Taking this approach, we mine the spatio-temporal rules on the data derived from the evolution of the training system and then check how well these rules describe spatio-temporal relations from the test system.

Classification algorithm

In this research we use a C4.5 ([303]) unpruned decision tree algorithm to mine spatio-temporal rules. It produces a set of non-conflicting rules (see Section 4.1.4). Each clause in the rules that are thus produced matches Definition 47. To be more precise, it takes the form (T, A, s, N), where each N

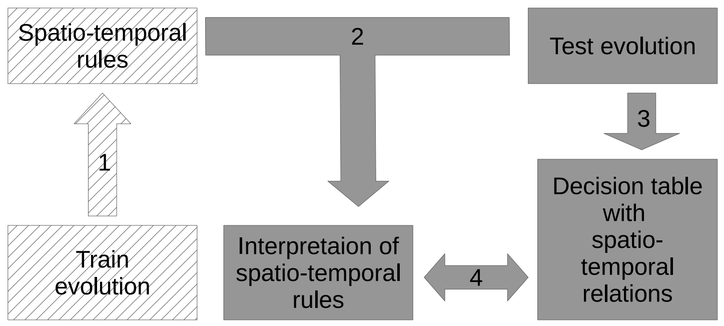
Spatio-temporal rules
2
Test evolution
1
3
Decision table with spatio-temporal relations
Train evolution
Interpretaion of spatio-temporal rules
4

Figure 6.4: Conceptual presentation of the validation of detection framework: 1 – spatio-temporal rules mining, 2 – interpretation of the rules in test evolution, 3 – finding occurrences of pattern instances and their spatio-temporal relations 4 – verification of prediction quality.

can be either: 1. $N = \{n \in \mathbb{N} : n \leqslant n_{thresh}\}$ or 2. $N = \{n \in \mathbb{N} : n > n_{thresh}\}$, where n_{thresh} is a natural number. This means that such rules can be interpreted unambiguously according to definitions given in Section 6.6.1 in any subevolution of any software.

Prediction of occurrences of design anti-pattern

Facts 30 and 31 provide an equivalence between appropriate spatio-temporal relations, which are used to construct the decision table and interpretation of spatio-temporal rules given in Definitions 45-47. In practice, this means that if spatio-temporal rules have good prediction quality on the test decision table, their interpretation accurately aims at areas where corresponding design anti-patterns occur in the test evolution. In particular, it can be used to predict where certain design anti-pattern may occur in the future, which is a potential application of the proposed framework in automated tools[12] which support the software development process.

The theory outlined earlier in this chapter, specifically Theorems 1, 3 and 6, allow us to state that spatio-temporal rules can be efficiently computed along with the evolution of the system in an adaptive manner. At a given revision r we can take spatio-temporal rules mined from the subevolution that ends at r, compute its interpretation and then select these spatio-temporal areas that cover revisions after r. Another way of looking at the matter is that

[12]For example tools built into Integrated Development Environment.

the revision r, which separates training data from test data, slides forward along with new commits. The training data is a sub-evolution before r and test data is the sub-evolution that starts with r. In such a setting, if we consider r to be the present moment, the test data can be perceived as the future evolution of the system.

6.7.3 The quality of spatio-temporal prediction

The proposed framework was empirically validated on the data derived from the evolution of systems listed in Section 6.7.1. Every dataset was used as a training evolution and tested against all other datasets. When the same dataset was used as training and test data, the training evolution was built from the first 70% revisions, and the test evolution – from the last 30% of revisions of the entire system history. This was done to simulate the detection of future occurrences of design anti-patterns, according to the description from the preceding paragraph.

The framework was tested in many configurations summarized in Table 6.3. Detailed results are given in Appendix A.6.1.

As we can see, the results of identification of individual types of design anti-patterns, have very high precision and at least moderate recall. Outstandingly good results are reported for AnemicEntity. We can hypothesize that this is related to the fact that AnemicEntity appears to be a *repellent* static pattern (see Section 6.7.4 below). The average F1 score for prediction of a single type of a design anti-pattern is 0.81.

The last two rows of Table 6.3 compare two configurations: 1. where all types of design anti-pattern are equated, which means that the algorithm has to predict areas where some design anti-patterns are, without telling their type, and 2. where each type of design anti-pattern was a separate decision class and the algorithm has to discern spatio-temporal areas with their occurrences. We can see that in the latter case the prediction quality is significantly worse. However, in terms of F1-score, the quality of prediction when all types of design-anti patterns were equated is 0.84, which makes it better than the average prediction of individual types.

6.7.4 Applicability to other domains

The preceding sections described the formal framework for encoding spatio-temporal relations in the evolution of the system, mining the spatio-temporal

Configuration	Avg. precision	Avg. recall	reference table
DataClump	1,00	0,75	A.13
SwissArmyKnife	0,84	0,57	A.14
AnemicEntity	1,00	0,94	A.15
BaseBean	1,00	0,74	A.16
Blob	0,99	0,51	A.17
YoYo	1,00	0,65	A.18
CircularDependency	0,98	0,58	A.19
BrainClass	1,00	0,81	A.20
any-pattern	0,94	0,72	A.21
differentiate	0,92	0,39	A.22

Table 6.3: The average quality of prediction of spatio temporal areas where occurrences of certain design patterns may appear. Rows with the names of the anti-pattern relate to the experiment where the model was trained to predict only this single type. The configuration called "any-pattern" corresponds to the experiment where all types of design-patterns were equated. The configuration called "differentiate" corresponds to the experiments where each type of a design-pattern was a separate decision class.

rules and assessing their predictive quality in the matter of predicting occurrences of popular design anti-patterns. Yet, the proposed method is more versatile and with some modifications can be applied to other tasks as well. The following sections describe exemplary applications and conclusions from the experiments described in this thesis.

Good and poor predictors of spatio-temporal patterns

In some configurations of the experiments described in Appendix A.6, the training and test data came from two different software systems. We can observe that the prediction quality is generally significantly better when we use certain systems to train the model. We can call these systems *good predictors*. By analogy: systems, whose use in this manner produces poor prediction quality can be called *poor predictors* ([369] discusses similar concepts). If we take into account the configuration with the best F1 measure described in the preceding paragraph, we can observe that: Wildfly and ArgoUML are good predictors (average F1=0.95) and JHotdraw is a bad predictor (F1=0.62).

In the case of systems like [7], this can be expected, since the available evolution of the system is relatively short (see Section 6.7.1), but in the case of other systems this phenomenon might be connected to the type of the system or the specific development process. The nature of this matter appears to be an interesting problem for further research.

Applicability to defect prediction

Please recall from Section 5.2.8 that defect prediction is a task in mining software repositories whose goal is to predict the number and the location of defects in the system source code. According to (e.g. [166], [167]), a dependency to an anti-pattern instance makes a class more likely to contain a bug. In terms of this thesis (see Section 6.5.2) this can be rephrased: entities from neighborhoods of instances of design anti-patterns are more likely to have a bug than those from their faraways. This view is based solely on spatial relations in the code. The question arises if spatio-temporal relations can be just as useful for defect prediction.

In Section 6.6.3 we have defined conditions that allow us to encode the spatio-temporal relations in the entire evolution between instances of design anti-patterns and other subgraphs. These are: 1. ability to uniquely identify the subgraph in each revision of the evolution 2. ability to define its lifespan. In the following paragraphs we show how these two notions can be defined for occurrences of defects in the software system. Conceptually, we assume that each defect has only a single occurrence that starts at the revision in which the defect was introduced and ends at the revision that actually fixes the problem. Please recall from Sections 2.2.1, 2.2.2 and 2.2.4 that in the software development process we may identify which commits are actual fixes of a defect in the software (informally: *bugfixes*) and which files are modified in it. We can heuristically assume that such a bugfix is always the final revision in the occurrence of a defect. The method of finding the initial revision when a defect occurred in the source code[13] may be based on more complex assumptions briefly described in the following paragraphs. They also describe different perspectives on the matter of defining which code entities constitute $SSn_?$ subgraph that represents the bug.

Single revision lifespan In arguably the most simplified approach we may consider that each bugfix is a sole isolated occurrence of a defect, whose

[13]It is the problem of bug-origin described in Section 5.2.8.

lifespan is limited to only a single revision. In such a setting the problem of unique identification of such a subgraph is trivial, as it is only present in a single revision and it may be identified by the set of all entities which are modified in the revision. This approach may be enhanced by splitting such a subgraph into its connected components, and treating each as a separate instance of the bug. Again, the identity of such a connected component is given by the set of all its nodes.

Heuristic identification of lifespan In the approach described in the previous paragraph the temporal range of the defect is related to the actual timing of the bugfix rather than to the entire presence of the defect in the source code of the system. To overcome this limitation one needs to find out when the specific bug was actually introduced. Certain heuristics mentioned in Section 5.2.8 can be helpful in this respect. A method proposed in [311], with additional improvements described in [194], identifies the origin of the bug by (simplifying a bit) finding a commit that introduced or changed the same lines in the source code that were modified in the bugfix commit. There are many variants of this method (see [299]), some of them also related to the structure of the source code, such as software snapshot, rather than to its textual contents (see [95]). Therefore, we can heuristically identify the bug-introducing commit by finding the revision which changes the same entities that are modified in the bugfix. In such a setting the occurrence of a defect consist of revision between bug-introducing commit and the bugfix.

Improved static patterns detection

Section 6.4 describes purely static methods that identify instances of certain design anti-patterns in a given software snapshot. Such a software snapshot is always part of broader software evolution and we can mine certain spatio-temporal rules in this software evolution, as described in Section 6.6. From the interpretation of these rules (see Section 6.6.1) for a given revision r and any type of pattern T, we can derive two "areas" of the SSn_r graph: Where instances of T are likely to be present, and where they are unlikely to be present. Consequently, we can use spatio-temporal rules to improve the identification quality of any static classifier.

Static detectors described in Section 6.4 can tell if a given subgraph is an instance of a specific design anti-pattern, but spatio-temporal rules, described in Section 6.6.4, can tell if this subgraph is in a spatio-temporal area, where one can expect an instance of the same design anti-pattern. These two outputs can be combined into a single compound classifier. Formally: Let C be a static classifier of type T, as defined in Definition 30, $R^+ = \left\{R_1^+, \ldots, R_n^+\right\}$ be a set of spatio-temporal rules with decision T, $R^- = \left\{R_1^-, \ldots, R_n^-\right\}$ be a set of spatio-temporal rules with decision NOT_T, SSn_r be a software snapshot at revision r, which is an element of software evolution Ev and let $G \in \mathscr{P}(SSn_r)$.

We will say that G is covered by a spatio-temporal rule R_i at revision r if there are two revisions $r_{start} \leqslant r \leqslant r_{end}$ such that all nodes of G are present in all revisions in the interval (r_{start}, r_{end}) and occurrence $(G, (r_{start}, r_{end}))$ is covered by R_i. With this notation we can define the following compound classifiers for a fixed revision r:

$$C^{+R^+}(G) = \begin{cases} 1 & \text{if } C(G) = 1 \text{ and } G \text{ is covered by a rule from } R^+ \\ 0 & \text{otherwise} \end{cases} \tag{6.15}$$

$$C^{-R^-}(G) = \begin{cases} 1 & \text{if } C(G) = 1 \text{ and } G \text{ is covered by none of the rules from } R^- \\ 0 & \text{otherwise} \end{cases} \tag{6.16}$$

Conceptually, $C^{+R^+}(G)$ is a classifier that considers a given graph to be an instance of a specific pattern only if both conditions hold: it has a specific static structure defined by C and it appears in a specific spatio-temporal area defined by positive rules from R^+.
Analogously: $C^{-R^-}(G)$ is a classifier that considers a given graph to be an instance of a specific pattern only if both conditions hold: it has a specific static structure defined by C and it appears outside of specific spatio-temporal area defined by negative rules from R^-.

We can further combine the following two classifiers into another one:

$$C^{-R^-+R^+}(G) = \begin{cases} 1 & \text{if } C^{-R^-}(G) = 1 \text{ and } C^{-R^-}(G) = 1 \\ 0 & \text{otherwise} \end{cases} \tag{6.17}$$

Conceptually, this combines the static classifier C with the knowledge about both positive and negative spatio-temporal rules in software evolution. Clearly, such a compound classifier can reduce the number of false positives but also increase the number of false negatives, thus it does not necessarily improve the quality of static classification. In practice, it appears that such a construct improves the classification quality by an average of 4% in terms of $F1$ measure, if we mine the spatio-temporal rules from the very beginning of the software evolution and use them to identify static patterns in a separate, final period of this evolution. Details of a respective experiment is given in Appendix A.6.2

Attractors and repellents

The decision table described in Section 6.6.2 provides information about all spatio-temporal relations in software evolution. In particular, we can use it to derive the number of pairs of close occurrences of pattern instances. Formally, for any two T_1, T_2 such that T_1 and T_2 are types of design anti-pattern, we can tell from the decision table the total number of pairs (o_1, o_2) such that o_1 is the occurrence of a pattern instance of type T_1 and o_2 is the occurrence of a pattern instance of type T_2 and these pattern instances are close. We will call each such pair an *attracting premise* for (T_1, T_2). Analogously we can tell from the decision table the number of pairs $(o_1^{'}, o_2^{'})$ such that $o_1^{'}$ is the occurrence of a pattern instance of type T_1 and $o_2^{'}$ is the occurrence of a pattern instance of type T_2 and these pattern instances are remote. We will call all such pairs *repelling premise* for (T_1, T_2). If the number of attracting premises is significantly higher than the number of repelling premises for some (T_1, T_2), that these types "attract" each other, so that it is more likely to find an instance of T_1 close to T_2 or T_2 close to T_1. If the number of repelling premises is significantly higher, we can expect that these types "repel" each other. To measure this in a formal manner, we introduce the concept of attraction ratio:

Definition 49. *Let T_1 and T_2 be two different types of design anti-patterns, $Attract_{(T_1,T_2)}$ denote the number of attracting premisses between T_1 and T_2, $Repell_{(T_1,T_2)}$ denote the number of repelling premisses between T_1 and T_2.*

The Attraction ratio *of T_1 and T_2 is*

$$\frac{Repell_{(T_1,T_2)} - Attract_{(T_1,T_2)}}{Attract_{(T_1,T_2)} + Repell_{(T_1,T_2)}}$$

The attraction ratio measures how biased a given pair of design anti-patterns is towards the attracting or repelling relation. Clearly, the value of this ratio can span form -1 (if we only derived attracting premisses) to +1 (if we only derived repelling premises). The value of it is 0 if the number of both types of premisses is equal.

As experimentally validated, Anemic Entity is the most repelling type of a design anti-pattern, as it has attraction ratio greater than 0.6 with three other types, and with all other types – the value is around 0.2. YoYo seems to be generally the most attracting type, but with only one exception of the pair (YoYo, CircularDependency), which is the pair with the greatest attracting ratio observed (0.78). For details, please refer to Experiments A.7.

6.8 Resume - overall pattern detection framework

This chapter describes the overall framework of mining and using spatio-temporal rules in the software development process, which is a key contribution of the current thesis.

- Section 6.1 provides information about the formal model of representation of the software structure and its evolution.
- Section 6.2 explains how this model can be built in an effective adaptive manner from data available in the actual systems such as SCM.
- Section 6.3 provides a formalism for the detection of *static patterns*, such as design anti-patterns. A formal framework for expressing the complexity of such detection is also provided as well as an efficient adaptive method of finding static patterns in the course of software evolution.
- Section 6.4 describes detection methods for a few popular design anti-patterns together with their complexity, both expressed in a formal manner described in Section 6.3.
- Section 6.5 explains how a single *static pattern* can be viewed in time, along with the evolution of the software. Based on that, concepts of *spatial relations* between two static patterns is formally introduced, namely: closeness and remoteness (see Sections 6.5.1 - 6.5.3).
  Also *temporal relations* between static patterns are formally defined

and a combination of the two: the *spatio-temporal relation* between patterns, which is one of key concepts in this thesis (Sections 6.5.4 - 6.5.5).

- Sections 6.5.6 - 6.5.7 describe in a formal manner how spatial, temporal and spatio-temporal relations can be computed in an efficient adaptive manner along with software development.
- Sections 6.6.1 and 6.7.3 define spatio-temporal rules and their interpretation in a formal manner and provide means to assess their quality in given software evolution.
- Sections 6.6.2 - 6.6.4 describe an efficient adaptive method of mining *spatio-temporal rules* in the course of software evolution.
- Sections 6.7 - 6.7.4 describe how the framework for mining spatio-temporal rules was empirically validated to detect occurrences of design anti-patterns and provide a few other examples of its other uses.

Chapter 7

Conclusions

The thesis presents a formal framework that enables us to gather, model and predict spatio-temporal phenomena in the life of a system under development. It starts with the representation of the structure of the source code of a system in the form of a property multigraph (see Section 6.1). Section 6.2.1 describes some simplifications which make the graph-theoretical model different from the respective semantic of the programming language. They make certain computations easier and, as experimentally validated in experiments described in Appendix A.1, they produce insignificant error in practical applications.

The model also includes a formal method for identifying instances of design anti-patterns in software, which are also described in graph-theoretical terms. In general, this can be as hard as the SUBGRAPH-ISOMORPHISM problem (See Section 6.3.3). However, incorporation of some domain expert knowledge in the field of software engineering described in Section 6.3.5, allows us to build a heuristic that appears to be very efficient in the practical application of mining a few popular design anti-patterns. This statement has been empirically confirmed by experiments described in Appendix A.5.2.

Section 6.3.4 introduces a formal measure of complexity of a static pattern detection, which is based on the number of nodes that have to be visited to tell if a given subgraph is an instance of that pattern. A generalized version of this problem is equivalent to regular navigational query in property graphs ([291]), but again the use of domain knowledge allows us to produce an efficient heuristic in the specific application of mining design anti-patterns. Its effectiveness has been empirically validated by experiments described in Appendix A.4.

One of the applications of this research is a set of classifiers for a few popular design anti-patterns, defined in the aforementioned formalism and described in Section 6.4. Their effectiveness has been empirically validated and compared to other published solutions in experiments described in Appendix A.5.1.

The fundamental concept of this thesis is a formal framework for describing and mining spatio-temporal relations between different instances of design anti-patterns. If comprises two different notions: closeness of pattern instances described in Section 6.5.2[1] and temporal relations between these instances described in Section 6.5.5. These concepts are then further used to introduce the notion of spatio-temporal relations and spatio-temporal rules, described in Sections 6.5.6 - 6.6.1. The above leads to the key contribution of this thesis: a method of predicting where and when instances of certain design anti-patterns can occur in the entire evolution of the software system, described in Sections 6.7.2-6.7.3. The experiments described in Appendix A.6 yield an empirical proof that the proposed method is capable of predicting such areas.

The framework can also be used for other applications such as defect prediction or improved, multi-aspect static pattern detection described in Section 6.7.4. In particular, the quality of static pattern detection can be improved on average by 4% in terms of F1 measure, when we take the mined spatio-temporal rules into account. This statement has been empirically confirmed in the experiment described in Appendix A.6.2.

The spatio-temporal rules describe the knowledge about spatio-temporal relations in software development. This model seems to be quite precise, as evidence from experiments described in Appendix A.6 suggest. Thus, we can analyze the structure of these rules, or the decision table they were mined from (see description in Sections 6.6.2-6.6.3) and use it to derive new concepts. *Attractor* and *repellent* design anti-patterns described in Section 6.7.4 are examples of such concepts.

Much attention in this thesis is paid to adaptivity of the proposed algorithms, understood in such a way that, after every commit, only a little computation is needed to have up-to-date data about spatio-temporal relations and spatio-temporal rules in software evolution. The adaptivity concepts are embedded in: 1. Adaptive construction of software snapshot after each commit (see Section 6.2.3 and Theorem 1 specifically) 2. Adaptive mining instances

[1]In fact two different types of closeness are defined.

of design anti-patterns (see Section 6.3.5 and Theorem 3 specifically). 3. Adaptive evaluation of spatial relations between pattern instances (see Section 6.5.6). 4. Adaptive evaluation of spatio-temporal relations (see Section 6.5.7 and Theorem 6 specifically). The above adaptive solutions, together with locality properties, whose nature is shown in experiments described in Appendix A.2, yield effective heuristic methods for mining spatio-temporal rules in the software development process.

7.1 Verification of hypotheses and goals

We can now conclude with a reference to the research hypotheses and goals outlined in Chapter 3. Please recall their statements:

- **H1: There are statistically significant temporal patterns in the software development process that can be used to predict the appearance of anti-patterns**
- **H2: Incorporation of expert knowledge can produce more efficient data mining algorithms in the domain of software development process**
- **G1: Formal model of design (anti-)patterns, code smells, and their evolution**
- **G2: Approximate model to represent static patterns in software systems**
- **G3: A model to represent temporal patterns in the software development process**
- **G4: Efficient mining algorithm especially fitted to the domain of software development process**

The experiments described in Appendix A.6 prove that we can efficiently predict where and when certain instances of design anti-patterns will be found, with an average of 0.81 F1 score (see Appendix A.6). Two simplifying assumptions are necessary to obtain such a result: Firstly, we cannot precisely predict graphs that correspond to actual pattern instances, in fact, we can only predict areas (i.e. broader graphs) that are 0-overlapping-close to them (see Section 6.5.1). Secondly, in some experiments, we have to group certain types of design anti-pattern (see Section 6.6.3 and Appendix A.6 for details). In other words, we cannot predict if a certain single type of pattern is likely to appear in a given area, but we can predict that an instance of one of a few types of a design anti-patterns is likely to appear in it. With the above disclaimers we can consider H1 hypothesis to be confirmed.

In purely theoretical terms, some problems described in this research are as hard as SUBGRAPH-ISOMORPHISM (see Section 6.3.3) or regular navigational queries in graphs (see Section 6.3.4). However, some specific simplifications based on expert knowledge from the domain of software design and engineering produce efficient heuristics. These include the analysis of the locality properties in the software development process (see Section 6.2.3 and Appendix A.2), the upper bounds on the length of paths in the SSn graph (see Section 6.3.4 and Appendix 30, A.4.3) and the use of graph indices based on software metrics (see Section 6.3.5 and Appendix A.5.2). All provide an efficient heuristic to mine pattern instances in software along its development. This allows us to state that hypothesis H2 is confirmed. Adding the adaptivity concepts referenced earlier in this chapter and described in detail in Sections 6.2.3, 6.3.5, 6.5.6 and 6.5.7, which are also founded on the software engineering domain knowledge, allows us to state that goal G4 is also achieved.

Section 6.3 provides a formal model for static patterns, including code smells and design anti-patterns. Section 6.4 contains definable patterns of a few popular design anti-patterns. Sections 6.5.4 - 6.5.5 define a formal model for the lifespan of static patterns and their mutual spatio-temporal relations. Given the above, we can conclude that goals G1 and G3 are achieved.

The approximate nature of the model is introduced in a twofold manner in this thesis: First - there is a formal graph-theoretical model described in Section 6.1 with intended simplifications described in Section 6.2.1. Second - there is an explicit approximate formalism of definable patterns described in Section 6.3.2. Consequently, goal G2 is clearly achieved.

7.2 Future Work

This section presents potential extensions of the framework presented in this thesis and possible further research.

7.2.1 Different sources of data

Raw data used in this research comes from publicly available SCMs and issue trackers of a few open-source, community-implemented software systems created in Java programming language (see Section 6.7.1). All these assumptions could be changed: Firstly, one can use the evolution of software that is not open-source and is developed by small commercial teams. There is evidence that such software tends to evolve differently (see [51], [112], [119], [132], [208], [273]). Secondly, one could use data from more software-development-supporting tools than just SCMs and issue trackers. These include e.g. test reports ([239]), messages ([84]) or even activities of particular developers ([286], [262]). Lastly, our framework, which is designed specifically to model the structure of a program written in Java (see Section 6.1), can be applied to software created in another programming language with some necessary modifications. Arguably, these modifications would be insignificant in the case of object-oriented languages. Adaptation to other paradigms (e.g. functional languages) would require significant reworking of the software snapshot structure.

7.2.2 Advanced model for pattern instance

In simple terms, the model of the structure of the software system used in this thesis is a multigraph and design patterns are represented as its specific subgraphs (see Section 6.1.1). The formalism for mining the design patterns is based on the assumption that we only consider the structure of the graph, i.e. the names of the nodes are ignored (see Section 6.3.2). In other words, the model is based only on the *structural properties* of the graphs. This assumption simplifies the formal framework, while keeping the quality comparable to current state-of-the-art algorithms in the domain of design anti-pattern detection (see Section 6.4.8 and Appendix A.5.1). Yet, there is evidence that information embedded in the names of code entities (*lexical properties*) can be used to improve the accuracy of detection (see [188], [192], [214], [243]).

The concept of the lifespan of a pattern instance, described in Section 6.5.4, is founded on the assumption that each instance of the design anti-pattern can be identified at different revisions of software evolution. Technically, it is based on the uniqueness of the name of the main entity between different revisions. This makes it vulnerable to e.g. changes of the name of an entity in a certain revision. There are heuristics available which cope with that problem, and many of them require both structural and lexical properties of the source code (see [349], [339], [350]). A potential further development of the framework proposed in this thesis is related to embedding in it lexical properties and to creating a more elaborate concept of the lifespan of a pattern instance.

7.2.3 Different view on time in software development process

Non-linear time

In this study it is assumed that the development of the system is done in a single branch only and that all commits can be ordered linearly. This is a key assumption, which allows us to build adaptive algorithms of mining spatio-temporal relations (see Section 6.3.5, 6.5.6 and 6.5.7). In fact, software is sometimes developed in multiple, parallel branches, and commits in these branches may interleave over time dimension (see [106]). An extension of the proposed framework so that it can cover a non-linear development process is a possible area for further research.

Time resolution

Another possible modification is related to the resolution of time in the analyzed process. We have assumed that software evolution is a sequence of software snapshots indexed by all revisions ordered linearly. Then, we have analyzed changes between each revision. However, in the model and the theorems described in Chapter 6 it is not necessary for the revisions to be subsequent, it is sufficient that the later is after the former. It means that we may look at changes in the software that are made over arbitrary units of time, provided that each contains at least one revision. Furthermore, this may potentially be extended to some process mining techniques (see Section 5.1.1), such as hierarchical mining. This would be possible if we could partition the software evolution into subsequent chunks, where each chunk consisted of a group of subsequent revisions, then mine spatio-temporal relations on each chunk individually, and then aggregate them on the level of all the chunks.

Definition of temporal relation

Allen's algebra, described in Section 4.3.1, is a helpful formalism, but it can arguably be too simplifying when it is used to model temporal relations between intervals of revisions in the software development process. For example, it cannot measure temporal proximity between intervals. Please note that relation between the separated intervals of revisions are indiscernible in terms of Allens theory in two cases: when they are separated by a single commit and when they are separated by thousands of commits. Thus Allen's algebra could be replaced by alternative formalism, which would incorporate more accurate model of temporal relations.

Sequential patterns

The temporal relation described in Section 6.5.5 enables us to express a relation between only two occurrences of static patterns. Arguably, some of the techniques of process mining described in Section 5.1.1, such as sequential pattern mining algorithms, may be used to extend the proposed framework so that it can also capture more sophisticated temporal phenomena in the software development process.

7.2.4 Local concepts of closeness and remoteness

We have assumed in this research that two code entities are close if they were close in any revision in the past. Similarly, two entities are remote, if there is no revision in which they were close (see Section 6.5.1). This allows us to construct efficient adaptive algorithms described in Section 6.5.6. In a sense it makes the relation of closeness and remoteness global. This assumption might be right for short-lived or not intensively developed software systems, but it might be oversimplifying in the case of very long software evolutions. To minimize this effect, one could change the notion of closeness to make it more "local in time". This could potentially be done with the use of a *sliding window* described in Section 5.1.3.

7.2.5 Rough-software-pattern

In order to define the notion of a *random-equivalent* (see Section 6.6.3), we have used concepts of *definable* and *indefinable patterns*, *super-* and *sub-approximation* and *upper bound* and *lower bound* described in Section 6.3.2.

They are clearly inspired by the rough sets theory (see [271]). Unlike the original, however, they are herein derived from the structural indiscernibility of graphs rather than from the indiscernibility derived from the actual information system. Since the said graphs represent the structure of the source code of a software system, we can introduce a generalized concept of *rough software pattern*, which itself is arguably an independent area of future research.

7.2.6 Adaptive mining of spatio-temporal rules

In the proposed framework, the spatio-temporal rules are mined in a quasi-adaptive manner: the actual decision table that describes spatio-temporal relations is built adaptively (see Section 6.6.4) and then a rule-based algorithm is run on it. This method is efficient, since the decision table is small compared to the original data from software evolution. However, the method can be extended in such a way that actual spatio-temporal rules are also computed in a fully adaptive manner. This should be possible as the facts about new spatio-temporal relations are constructed adaptively in each revision in an iterative manner (see Sections 6.5.7, 6.5.6 and 6.6.4). This means that these facts can be put into a stream of data, and then we can probably use an algorithm of mining rules from data streams (see [101], [223], [87]).

Bibliography

[1] Understanding Class Evolution in Object-Oriented Software. In *Proceedings of the 12th IEEE International Workshop on Program Comprehension*, Washington, DC, USA, 2004. IEEE Computer Society.

[2] Git, `http://git-scm.com/`, Jul 2011.

[3] Jira, `http://www.atlassian.com/software/jira/`, Jul 2011.

[4] Subversion, `http://subversion.tigris.org/`, Jul 2011.

[5] Argouml, `http://argouml.tigris.org/issues/`, Sep 2015.

[6] Checkstyle, `http://checkstyle.sourceforge.net/config_metrics.html`, June 2016.

[7] Jhotdraw, `http://www.jhotdraw.org/`, Apr 2018.

[8] Apache software foundation issue tracker, `https://issues.apache.org`, Nov 2020.

[9] Apache software foundation scm, `http://svn.apache.org/repos/`, Nov 2020.

[10] Argouml issue tracker, `http://argouml.tigris.org/servlets/ProjectIssues`, Nov 2020.

[11] Argouml source code, `https://github.com/argouml-tigris-org/argouml`, Nov 2020.

[12] Elasticsearch source code, `https://github.com/elastic/elasticsearch`, Nov 2020.

[13] Elasticsearch, `https://www.elastic.co/`, Nov 2020.

[14] Jhotdraw source code, `https://github.com/wrandelshofer/jhotdraw`, Nov 2020.

[15] Lucene solr source code, `https://gitbox.apache.org/repos/asf/lucene-solr.git`, Nov 2020.

[16] Lucene solr, `https://solr.apache.org/`, Nov 2020.

[17] Struts, `https://struts.apache.org/`, Nov 2020.

[18] Wildfly issue tracker, `https://issues.jboss.org`, Nov 2020.

[19] Wildfly source code, `https://github.com/wildfly/wildfly`, Nov 2020.

[20] Wildfly, `https://www.wildfly.org/`, Nov 2020.

[21] Xerces, `https://xerces.apache.org/#xerces2-j`, Nov 2020.

[22] Bugzilla, `https://www.bugzilla.org/`, Sep 2021.

[23] P. Abate, R. Di Cosmo, J. Boender, and S. Zacchiroli. Strong dependencies between software components. In *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*, ESEM '09, pages 89–99, Washington, DC, USA, 2009. IEEE Computer Society.

[24] A. AbuHassan, M. Alshayeb, and L. Ghouti. Software smell detection techniques: A systematic literature review. *Journal of Software: Evolution and Process*, 33(3), Mar. 2021.

[25] C. C. Aggarwal and H. Wang. *Managing and Mining Graph Data*. Springer Publishing Company, Incorporated, 1st edition, 2010.

[26] K. K. Aggarwal, Y. Singh, A. Kaur, and R. Malhotra. Software Design Metrics for Object-Oriented Software.

[27] R. Agrawal, T. Imieliński, and A. Swami. Mining Association Rules Between Sets of Items in Large Databases. *SIGMOD Rec.*, 22(2):207–216, June 1993.

[28] R. Agrawal and R. Srikant. Fast Algorithms for Mining Association Rules in Large Databases. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, pages 487–499, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[29] R. Agrawal and R. Srikant. Mining sequential patterns. pages 3–14, Mar. 1995.

[30] J. F. Allen. Maintaining Knowledge About Temporal Intervals. *Commun. ACM*, 26(11):832–843, Nov. 1983.

[31] O. Alonso, P. T. Devanbu, and M. Gertz. Expertise identification and visualization from cvs. In *In Proceedings of the 2008 International Working Conference on Mining Software Repositories*, pages 125–128.

[32] S. Amasaki, T. Yoshitomi, O. Mizuno, Y. Takagi, and T. Kikuno. A New Challenge for Applying Time Series Metrics Data to Software Quality Estimation. *Software Quality Control*, 13(2):177–193, June 2005.

[33] R. Ambauen, S. Fischer, and H. Bunke. Graph Edit Distance with Node Splitting and Merging, and Its Application to Diatom Identification. In *Proceedings of the 4th IAPR International Conference on Graph Based Representations in Pattern Recognition*, GbRPR'03, pages 95–106, York, UK, 2003. Springer-Verlag.

[34] R. Angles, M. Arenas, P. Barceló, A. Hogan, J. Reutter, and D. Vrgoč. Foundations of Modern Query Languages for Graph Databases. *ACM Computing Surveys*, 50(5):1–40, Nov. 2017.

[35] G. Antoniol, M. Di Penta, and E. Merlo. Predicting Refactoring Activities via Time Series. In *The First International Workshop on Refactoring (REFACE*, 2003.

[36] F. Arcelli Fontana, M. V. Mäntylä, M. Zanoni, and A. Marino. Comparing and experimenting machine learning techniques for code smell detection. *Empir Software Eng*, 21(3):1143–1191, June 2016.

[37] F. Arcelli Fontana and M. Zanoni. A tool for design pattern detection and software architecture reconstruction. *Information Sciences*, 181(7):1306–1324, Apr. 2011.

[38] K. Arnold, J. Gosling, and D. Holmes. *The Java Programming Language, 4th Edition.* Addison-Wesley Professional, fourth edition, Aug. 2005.

[39] G. Atluri, A. Karpatne, and V. Kumar. Spatio-Temporal Data Mining: A Survey of Problems and Methods. *ACM Comput. Surv.*, 51(4):83:1–83:41, Aug. 2018.

[40] L. Aversano, G. Canfora, L. Cerulo, C. Del Grosso, and M. Di Penta. An empirical study on the evolution of design patterns. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE '07, pages 385–394, Dubrovnik, Croatia, 2007. ACM.

[41] L. Aversano, L. Cerulo, and C. Del Grosso. Learning from bug-introducing changes to prevent fault prone code. In *Ninth International Workshop on Principles of Software Evolution: In Conjunction with the 6th ESEC/FSE Joint Meeting*, IWPSE '07, pages 19–26, Dubrovnik, Croatia, 2007. ACM.

[42] K. Ayari, P. Meshkinfam, G. Antoniol, and M. Di Penta. Threats on building models from CVS and Bugzilla repositories: The Mozilla case study. In *Proceedings of the 2007 Conference of the Center for Advanced Studies on Collaborative Research*, CASCON '07, pages 215–228, Richmond Hill, Ontario, Canada, 2007. ACM.

[43] D. Baca, B. Carlsson, and L. Lundberg. Evaluating the cost reduction of static code analysis for software security. In *Proceedings of the Third ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, PLAS '08, pages 79–88, Tucson, AZ, USA, 2008. ACM.

[44] A. Bacchelli, M. D'Ambros, and M. Lanza. Are Popular Classes More Defect Prone? In D. Rosenblum and G. Taentzer, editors, *Fundamental Approaches to Software Engineering*, volume 6013 of *Lecture Notes in Computer Science*, pages 59–73. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2010.

[45] M. Balzer, O. Deussen, and C. Lewerentz. Voronoi treemaps for the visualization of software metrics. In *Proceedings of the 2005 ACM*

Symposium on Software Visualization, SoftVis '05, pages 165–172, St. Louis, Missouri, 2005. ACM.

[46] C. Bartoszuk, G. Timoszuk, R. Dabrowski, and K. Stencel. On Visual Assessment of Software Quality. *e-Informatica Software Engineering Journal*, 8(1):7–26, 2014.

[47] J. G. Bazan and M. Szczuka. The rough set exploration system. In *Transactions on Rough Sets III*, pages 37–56. Springer, 2005.

[48] M. L. Bernardi, M. Cimitile, and G. Di Lucca. Design Pattern Detection Using a DSL-driven Graph Matching Approach. *J. Softw. Evol. Process*, 26(12):1233–1266, Dec. 2014.

[49] J. Bevan and E. J. Whitehead. Identification of Software Instabilities. In *Proceedings of the 10th Working Conference on Reverse Engineering*, WCRE '03, Washington, DC, USA, 2003. IEEE Computer Society.

[50] J. Bevan, E. J. Whitehead, S. Kim, and M. Godfrey. Facilitating software evolution research with kenyon. *SIGSOFT Softw. Eng. Notes*, 30(5):177–186, Sept. 2005.

[51] J. M. Bieman, A. A. Andrews, and H. J. Yang. Understanding Change-Proneness in OO Software Through Visualization. In *Proceedings of the 11th IEEE International Workshop on Program Comprehension*, IWPC '03, Washington, DC, USA, 2003. IEEE Computer Society.

[52] J. M. Bieman and B.-K. Kang. Cohesion and reuse in an object-oriented system. *ACM SIGSOFT Software Engineering Notes*, 20(SI):259–262, Aug. 1995.

[53] D. Binkley. Source Code Analysis: A Road Map. In *2007 Future of Software Engineering*, FOSE '07, pages 104–119, Washington, DC, USA, 2007. IEEE Computer Society.

[54] A. Binun and G. Kniesel. DPJF - Design Pattern Detection with High Accuracy. In *2012 16th European Conference on Software Maintenance and Reengineering*, pages 245–254, Szeged, Hungary, Mar. 2012. IEEE.

[55] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. Devanbu. Fair and balanced?: Bias in bug-fix datasets. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, pages 121–130, Amsterdam, The Netherlands, 2009. ACM.

[56] C. Bird, P. C. Rigby, E. T. Barr, D. J. Hamilton, D. M. German, and P. Devanbu. The promises and perils of mining git. In *Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories*, volume 0 of *MSR '09*, pages 1–10, Washington, DC, USA, 2009. IEEE Computer Society.

[57] S. Boroday, A. Petrenko, J. Singh, and H. Hallal. Dynamic analysis of java applications for multithreaded antipatterns. In *Proceedings of the Third International Workshop on Dynamic Analysis*, WODA '05, pages 1–7, St. Louis, Missouri, 2005. ACM.

[58] J. C. Bose, W. M. P. van der Aalst, I. Žliobaite, and M. Pechenizkiy. Handling Concept Drift in Process Mining. In *Proceedings of the 23rd International Conference on Advanced Information Systems Engineering*, CAiSE'11, pages 391–405, London, UK, 2011. Springer-Verlag.

[59] M. Bramer. Introduction to Data Mining. In M. Bramer, editor, *Principles of Data Mining*, Undergraduate Topics in Computer Science, pages 1–8. Springer, London, 2013.

[60] S. Breu and T. Zimmermann. Mining Aspects from Version History. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, pages 221–230, Tokyo, Nov. 2006. IEEE Computer Society.

[61] C. Brindescu, M. Codoban, S. Shmarkatiuk, and D. Dig. How do centralized and distributed version control systems impact software changes? In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 322–333, New York, NY, USA, May 2014. Association for Computing Machinery.

[62] F. P. Brooks. No Silver Bullet Essence and Accidents of Software Engineering. *Computer*, 20(4):10–19, Apr. 1987.

[63] F. P. Brooks. *The Mythical Man-month (Anniversary Ed.)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[64] W. H. Brown, R. C. Malveau, H. W. "Skip" McCormick, and T. J. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley &amp; Sons, Inc., New York, NY, USA, 1st edition, 1998.

[65] I. I. Brudaru and A. Zeller. What is the long-term impact of changes? In *Proceedings of the 2008 International Workshop on Recommendation Systems for Software Engineering*, RSSE '08, pages 30–32, Atlanta, Georgia, 2008. ACM.

[66] H. Bunke and G. Allermann. Inexact Graph Matching for Structural Pattern Recognition. *Pattern Recogn. Lett.*, 1(4):245–253, May 1983.

[67] R. P. L. Buse and W. R. Weimer. A Metric for Software Readability. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA '08, pages 121–130, Seattle, WA, USA, 2008. ACM.

[68] R. P. L. Buse and W. R. Weimer. Automatically documenting program changes. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pages 33–42, Antwerp, Belgium, 2010. ACM.

[69] J. Businge, A. Serebrenik, and M. van den Brand. An empirical study of the evolution of Eclipse third-party plug-ins. In *Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE)*, IWPSE-EVOL '10, pages 63–72, Antwerp, Belgium, 2010. ACM.

[70] T. Caelli and S. Kosinov. An eigenspace projection clustering method for inexact graph matching. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26(4):515–519, Apr. 2004.

[71] X. Cai, Y. Niu, S. Geng, J. Zhang, Z. Cui, J. Li, and J. Chen. An under-sampled software defect prediction method based on hybrid multi-objective cuckoo search. *Concurrency and Computation: Practice and Experience*, 32(5), Mar. 2020.

[72] G. Canfora and L. Cerulo. Fine grained indexing of software repositories to support impact analysis. In *Proceedings of the 2006 International Workshop on Mining Software Repositories*, MSR '06, pages 105–111, Shanghai, China, 2006. ACM.

[73] J. G. Carbonell, R. S. Michalski, and T. M. Mitchell. An Overview Of Machine Learning. 1983.

[74] R. Caruana and A. Niculescu-mizil. An Empirical Comparison of Supervised Learning Algorithms. In *In Proc. 23rd International Conference on Machine Learning*, pages 161–168, 2006.

[75] J. T. Chargo. *Automated Software Architecture Extraction Using Graph-based Clustering*. PhD thesis, Iowa State University, 2013.

[76] Y. Chen, M. Thurley, and M. Weyer. Understanding the Complexity of Induced Subgraph Isomorphisms. In L. Aceto, I. Damgård, L. Goldberg, M. Halldórsson, A. Ingólfsdóttir, and I. Walukiewicz, editors, *Automata, Languages and Programming*, volume 5125 of *Lecture Notes in Computer Science*, pages 587–596. Springer Berlin Heidelberg, 2008.

[77] Y. Chi, H. Wang, P. S. Yu, and R. R. Muntz. Moment: Maintaining closed frequent itemsets over a stream sliding window. In *Data Mining, 2004. ICDM Fourth IEEE International Conference on*, pages 59–66. IEEE, Nov. 2004.

[78] S. R. Chidamber and C. F. Kemerer. A Metrics Suite for Object Oriented Design. *IEEE Trans. Softw. Eng.*, 20(6):476–493, June 1994.

[79] C. Collberg, S. Kobourov, J. Nagra, J. Pitts, and K. Wampler. A system for graph-based visualization of the evolution of software. In *Proceedings of the 2003 ACM Symposium on Software Visualization*, SoftVis '03, pages 77–ff, San Diego, California, 2003. ACM.

[80] J. E. Cook and A. L. Wolf. Discovering Models of Software Processes from Event-Based Data. *ACM Transactions on Software Engineering and Methodology*, 7:215–249, 1998.

[81] S. A. Cook. The Complexity of Theorem-proving Procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, pages 151–158, Shaker Heights, Ohio, USA, 1971. ACM.

[82] C. Cortes and V. Vapnik. Support-vector networks. *Machine Learning*, 20(3):273–297, Sept. 1995.

[83] J. Costa, F. Santana, and C. de Souza. Understanding Open Source Developers' Evolution Using TransFlow. In L. Carriço, N. Baloian, and B. Fonseca, editors, *Groupware: Design, Implementation, and Use*, volume 5784 of *Lecture Notes in Computer Science*, pages 65–78. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2009.

[84] D. Cubranic and G. Murphy. Hipikat: Recommending pertinent software development artifacts. In *25th International Conference on Software Engineering, 2003. Proceedings.*, pages 408–418, Portland, OR, USA, 2003. IEEE.

[85] D. Čubranić, G. C. Murphy, J. Singer, and K. S. Booth. Learning from project history: A case study for software development. In *Proceedings of the 2004 ACM Conference on Computer Supported Cooperative Work*, CSCW '04, pages 82–91, Chicago, Illinois, USA, 2004. ACM.

[86] D. A. da Costa, S. McIntosh, W. Shang, U. Kulesza, R. Coelho, and A. E. Hassan. A Framework for Evaluating the Results of the SZZ Approach for Identifying Bug-Introducing Changes. *IEEE Transactions on Software Engineering*, 43(7):641–657, July 2017.

[87] V. G. T. da Costa, A. C. P. d. L. F. de Carvalho, and S. Barbon Junior. Strict Very Fast Decision Tree: A memory conservative algorithm for data stream mining. *Pattern Recognition Letters*, 116:22–28, Dec. 2018.

[88] R. Dąbrowski, K. Stencel, and G. Timoszuk. Software Is a Directed Multigraph. In I. Crnkovic, V. Gruhn, and M. Book, editors, *Software Architecture*, volume 6903, pages 360–369. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[89] R. Dabrowski, K. Stencel, and G. Timoszuk. Software is a directed multigraph (and so is software process). Mar. 2011.

[90] R. Dąbrowski, G. Timoszuk, and K. Stencel. One Graph to Rule Them All Software Measurement and Management. *Fundamenta Informaticae*, 128(1-2):47–63, 2013.

[91] M. D'Ambros and M. Lanza. Reverse Engineering with Logical Coupling. In *Proceedings of the 13th Working Conference on Reverse Engineering*, pages 189–198, Washington, DC, USA, 2006. IEEE Computer Society.

[92] M. DAmbros and M. Lanza. Distributed and Collaborative Software Evolution Analysis with Churrasco. *Sci. Comput. Program.*, 75(4):276–287, Apr. 2010.

[93] M. D'Ambros, M. Lanza, and R. Robbes. Evaluating defect prediction approaches: A benchmark and an extensive comparison. *Empirical Software Engineering*, 17(4-5):531–577, Aug. 2012.

[94] J. Davey and E. Burd. Clustering and concept analysis for software evolution. In *Proceedings of the 4th International Workshop on Principles of Software Evolution*, IWPSE '01, pages 146–149, Vienna, Austria, 2001. ACM.

[95] S. Davies, M. Roper, and M. Wood. Comparing text-based and dependence-based approaches for determining the origins of bugs. *Journal of Software: Evolution and Process*, 26, Jan. 2014.

[96] A. K. A. de Medeiros, W. M. P. van der Aalst, and A. J. M. M. Weijters. Workflow Mining: Current Status and Future Directions. In R. Meersman, Z. Tari, and D. Schmidt, editors, *On The Move to Meaningful Internet Systems 2003: CoopIS, DOA, and ODBASE*, volume 2888 of *Lecture Notes in Computer Science*, pages 389–406. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

[97] L. Devroye, L. Györfi, and G. Lugosi. *A Probabilistic Theory of Pattern Recognition.* Springer, 1996.

[98] S. Diehl. *Software Visualization: Visualizing the Structure, Behaviour, and Evolution of Software.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.

[99] S. Diehl. Visualizing the Evolution of Software Systems. In *Software Visualization*, pages 129–147. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[100] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson. Automated Detection of Refactorings in Evolving Components. In *Proceedings of the 20th European Conference on Object-Oriented Programming*, volume 4067 of *ECOOP'06*, pages 404–428, Nantes, France, 2006. Springer-Verlag.

[101] P. Domingos and G. Hulten. Mining high-speed data streams. In *Proceedings of the Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining - KDD '00*, pages 71–80, Boston, Massachusetts, United States, 2000. ACM Press.

[102] J. Dong, Y. Zhao, and T. Peng. A Review of Design Pattern Mining Techniques. *Int. J. Softw. Eng. Knowl. Eng.*, 2009.

[103] X. Dong and M. W. Godfrey. Identifying Architectural Change Patterns in Object-Oriented Systems. In *Proceedings of the 2008 The 16th IEEE International Conference on Program Comprehension*, ICPC '08, pages 33–42, Washington, DC, USA, 2008. IEEE Computer Society.

[104] J. Dougherty, R. Kohavi, and M. Sahami. Supervised and Unsupervised Discretization of Continuous Features. In *Machine Learning: Proceedings of the 12th International Conference*, pages 194–202, 1995.

[105] D. Draheim and L. Pekacki. Analytical Processing of Version Control Data: Towards a Process-Centric Viewpoint. 2003.

[106] V. Driessen. Gitflow, `https://datasift.github.io/gitflow/IntroducingGitFlow.html`, Jan 2010.

[107] F. Eichinger and K. Bohm. Software-Bug Localization with Graph Mining. In A. K. Elmagarmid, C. C. Aggarwal, and H. Wang, editors, *Managing and Mining Graph Data*, volume 40 of *Advances in Database Systems*, pages 515–546. Springer US, Boston, MA, 2010.

[108] S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mockus. Does Code Decay? Assessing the Evidence from Change Management Data. *IEEE Trans. Softw. Eng.*, 27(1):1–12, Jan. 2001.

[109] M. G. Elfeky, W. G. Aref, and A. K. Elmagarmid. Periodicity detection in time series databases. In *IEEE TRANS. KNOWL. DATA ENG*, 2005.

[110] D. Eppstein. Subgraph Isomorphism in Planar Graphs and Related Problems. *Journal of Graph Algorithms and Applications*, 3(3):1–27, 1999.

[111] M. A. Eshera and K.-S. Fu. A graph distance measure for image analysis. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-14(3):398–408, May 1984.

[112] J. Estublier, D. Leblang, A. van der Hoek, R. Conradi, G. Clemm, W. Tichy, and D. W. Weber. Impact of software engineering research on the practice of software configuration management. *ACM Trans. Softw. Eng. Methodol.*, 14(4):383–430, Oct. 2005.

[113] W. Evans, C. Fraser, and F. Ma. Clone detection via structural abstraction. *Software Quality Journal*, 17(4):309–330, Dec. 2009.

[114] E. Fernandes, J. Oliveira, G. Vale, T. Paiva, and E. Figueiredo. A review-based comparative study of bad smell detection tools. In *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering*, pages 1–12, Limerick Ireland, June 2016. ACM.

[115] J. Fernandez-Ramil, A. Lozano, M. Wermelinger, and A. Capiluppi. Empirical Studies of Open Source Evolution. In *Software Evolution*, pages 263–288. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[116] M. Fischer. *EvoZilla: Longitudinal Evolution Analysis of Large Scale Software Systems,*. PhD thesis, University of Zurich, Dec. 2006.

[117] M. Fischer, M. Pinzger, and H. Gall. Populating a Release History Database from Version Control and Bug Tracking Systems. In *Proceedings of the International Conference on Software Maintenance*, ICSM '03, Washington, DC, USA, 2003. IEEE Computer Society.

[118] R. A. Fisher. The Use of Multiple Measurements in Taxonomic Problems. *Annals of Eugenics*, 7(2):179–188, Sept. 1936.

[119] B. Fluri, E. Giger, and H. C. Gall. Discovering Patterns of Change Types. pages 463–466, L'Aquila, Italy, Sept. 2008.

[120] P. Foggia, C. Sansone, and M. Vento. A performance comparison of five algorithms for graph isomorphism. In *In Proceedings of the 3rd IAPR TC-15 Workshop on Graph-based Representations in Pattern Recognition*, pages 188–199, 2001.

[121] M. Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, first edition, Nov. 2002.

[122] M. Fowler and K. Beck. *Refactoring Improving the Design of Existing Code*. Addison-Wesley, first edition, July 2013.

[123] E. Frank. *Pruning Decision Trees and Lists*. PhD thesis, Department of Computer Science, University of Waikato, 2000.

[124] H. Gall, K. Hajek, and M. Jazayeri. Detection of Logical Coupling Based on Product Release History. In *Proceedings of the International Conference on Software Maintenance*, ICSM '98, pages 190+, Washington, DC, USA, 1998. IEEE Computer Society.

[125] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, first edition, Nov. 1994.

[126] X. Gao, B. Xiao, D. Tao, and X. Li. A survey of graph edit distance. *Pattern Analysis and Applications*, 13(1):113–129, Feb. 2010.

[127] C. d. S. Garcia, A. Meincheim, E. R. Faria Junior, M. R. Dallagassa, D. M. V. Sato, D. R. Carvalho, E. A. P. Santos, and E. E. Scalabrin. Process mining techniques and applications – A systematic mapping study. *Expert Systems with Applications*, 133:260–295, Nov. 2019.

[128] S. Garcia, J. Luengo, J. A. Sáez, V. López, and F. Herrera. A Survey of Discretization Techniques: Taxonomy and Empirical Analysis in Supervised Learning. *Knowledge and Data Engineering, IEEE Transactions on*, 25(4):734–750, Apr. 2013.

[129] T. Girba, S. Ducasse, and M. Lanza. Yesterday's weather: Guiding early reverse engineering efforts by summarizing the evolution of changes. pages 40–49, Chicago, IL, USA.

[130] T. Girba and M. Lanza. Visualizing and Characterizing the Evolution of Class Hierarchies. In *Fifth International Workshop on Object-Oriented Reengineering*, 2004.

[131] R. Giugno and D. Shasha. GraphGrep: A fast and universal method for querying graphs. In *Pattern Recognition, 2002. Proceedings. 16th International Conference On*, volume 2, pages 112–115 vol.2. IEEE, 2002.

[132] M. W. Godfrey and Q. Tu. Evolution in open source software: A case study. In *Software Maintenance, 2000. Proceedings. International Conference On*, pages 131–142. IEEE, 2000.

[133] B. Goel and Y. Singh. Empirical Investigation of Metrics for Fault Prediction on Object-Oriented Software. In R. Lee and H.-K. Kim, editors, *Computer and Information Science*, volume 131 of *Studies in Computational Intelligence*, pages 255–265. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2008.

[134] V. U. Gómez, A. Kellens, J. Brichau, and T. D'Hondt. Time warp, an approach for reasoning over system histories. In *Proceedings of the Joint International and Annual ERCIM Workshops on Principles of Software Evolution (IWPSE) and Software Evolution (Evol) Workshops*, IWPSE-Evol '09, pages 79–88, Amsterdam, The Netherlands, 2009. ACM.

[135] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy. Predicting fault incidence using software change history. *Software Engineering, IEEE Transactions on*, 26(7):653–661, July 2000.

[136] G. Greco, A. Guzzo, and L. Pontieri. Mining Hierarchies of Models: From Abstract Views to Concrete Specifications. In W. van der Aalst, B. Benatallah, F. Casati, and F. Curbera, editors, *Business Process Management*, volume 3649 of *Lecture Notes in Computer Science*, pages 32–47. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2005.

[137] G. Greco, A. Guzzo, L. Pontieri, and D. Saccà. Mining Expressive Process Models by Clustering Workflow Traces. In H. Dai, R. Srikant, and C. Zhang, editors, *Advances in Knowledge Discovery and Data*

Mining, volume 3056 of *Lecture Notes in Computer Science*, pages 52–62. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2004.

[138] O. Greevy. Analyzing feature traces to incorporate the semantics of change in software evolution analysis. In *In Proceedings of ICSM 2005 (21th International Conference on Software Maintenance*, pages 347–356, 2005.

[139] Y.-G. Guéhéneuc. A reverse engineering tool for precise class diagrams. In *2004 conference of the Centre for Advanced Studies on Collaborative research*, pages 28–41, Jan. 2004.

[140] Y.-G. Gueheneuc, J.-Y. Guyomarch, and H. Sahraoui. Improving design-pattern identification: A new approach and an exploratory study. *Software Quality Journal*, 18(1):145–174, Mar. 2010.

[141] V. Guralnik, D. Wijesekera, and J. Srivastava. Pattern Directed Mining of Sequence Data. In *Fourth International Conference on Knowledge Discovery and Data Mining*, pages 51–57, 1998.

[142] M. H. Halstead. *Elements of Software Science (Operating and Programming Systems Series)*. Elsevier Science Ltd, 1977.

[143] M. Harman, D. Binkley, K. Gallagher, N. Gold, and J. Krinke. Dependence clusters in source code. *ACM Trans. Program. Lang. Syst.*, 32(1):1–33, Nov. 2009.

[144] A. E. Hassan and R. C. Holt. Predicting change propagation in software systems. In *20th IEEE International Conference on Software Maintenance*, 2004.

[145] M. H. Hassoun. *Fundamentals of Artificial Neural Networks*. MIT Press, Cambridge, MA, USA, 1st edition, 1995.

[146] T. J. Hastie, R. J. Tibshirani, and J. H. Friedman. *The Elements of Statistical Learning : Data Mining, Inference, and Prediction*. Springer, 2nd ed. 2009. corr. 7th printing 2013 edition, Apr. 2017.

[147] J. Henkel and A. Diwan. CatchUp!: Capturing and Replaying Refactorings to Support API Evolution. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 274–283, St. Louis, MO, USA, 2005. ACM.

[148] M. Heričko and S. Beloglavec. A Composite Design-Pattern Identification Technique. *Informatica*, 29, 2005.

[149] I. Herraiz, J. M. Gonzalez Barahona, and G. Robles. Determinism and evolution. In *Proceedings of the 2008 International Working Conference on Mining Software Repositories*, MSR '08, pages 1–10, Leipzig, Germany, 2008. ACM.

[150] I. Herraiz, G. Robles, and J. M. Gonzalez Barahona. Towards Predictor Models for Large Libre Software Projects. In *Proceedings of the 2005 Workshop on Predictor Models in Software Engineering*, PROMISE '05, pages 1–6, St. Louis, Missouri, 2005. ACM.

[151] I. Herraiz, D. Rodriguez, G. Robles, and J. M. Gonzalez Barahona. The Evolution of the Laws of Software Evolution: A Discussion Based on a Systematic Literature Review. *ACM Comput. Surv.*, 46(2), Dec. 2013.

[152] K. S. Herzig. Capturing the long-term impact of changes. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ICSE '10, pages 393–396, Cape Town, South Africa, 2010. ACM.

[153] K. S. Herzig. *Mining and Untangling Change Genealogies*. PhD thesis, Universität des Saarlandes, 2012.

[154] D. Heuzeroth, T. Holl, G. Hogstrom, and W. Lowe. Automatic design pattern detection. In *Program Comprehension, 2003. 11th IEEE International Workshop On*, pages 94–103. IEEE, May 2003.

[155] Y. Higo, K. Murao, S. Kusumoto, and K. Inoue. Predicting fault-prone modules based on metrics transitions. In *Proceedings of the 2008 Workshop on Defects in Large Software Systems*, DEFECTS '08, pages 6–10, Seattle, Washington, 2008. ACM.

[156] E. M. Hill, C. Parnin, and A. P. Black. How we refactor, and how we know it. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 287–297, Washington, DC, USA, 2009. IEEE Computer Society.

[157] A. Hindle and D. M. German. SCQL: A Formal Model and a Query Language for Source Control Repositories. In *Proceedings of the 2005 International Workshop on Mining Software Repositories*, volume 30 of *MSR '05*, pages 1–5, St. Louis, Missouri, 2005. ACM.

[158] M. Hitz and B. Montazeri. Measuring coupling and cohesion in object-oriented systems. In *Proceedings of International Symposium on Applied Corporate Computing*, pages 25–27, 1995.

[159] J. E. Hopcroft and J. K. Wong. Linear Time Algorithm for Isomorphism of Planar Graphs (Preliminary Report). In *Proceedings of the Sixth Annual ACM Symposium on Theory of Computing*, STOC '74, pages 172–184, Seattle, Washington, USA, 1974. ACM.

[160] S. Hosseini, B. Turhan, and D. Gunarathna. A Systematic Literature Review and Meta-Analysis on Cross Project Defect Prediction. *IEEE Transactions on Software Engineering*, 45(2):111–147, Feb. 2019.

[161] D. Hovemeyer and W. Pugh. Finding Bugs is Easy. *SIGPLAN Not.*, 39(12):92–106, Dec. 2004.

[162] K.-Y. Huang and C.-H. Chang. Efficient mining of frequent episodes from complex sequences. *Information Systems*, 33(1):96–114, Mar. 2008.

[163] E. Hüllermeier and S. Vanderlooy. Combining predictions in pairwise classification: An optimal adaptive voting strategy and its relation to weighted voting. *Pattern Recognition*, 43(1):128–142, Jan. 2010.

[164] F. Iglesias and W. Kastner. Analysis of Similarity Measures in Times Series Clustering for the Discovery of Building Energy Patterns. *Energies*, 6(2):579–597, Jan. 2013.

[165] C. Izurieta and J. M. Bieman. A multiple case study of design pattern decay, grime, and rot in evolving software systems. *Software Quality Journal*, pages 1–35, Feb. 2012.

[166] F. Jaafar, Y. G. Gueheneuc, S. Hamel, and F. Khomh. Mining the relationship between anti-patterns dependencies and fault-proneness. In *Reverse Engineering (WCRE), 2013 20th Working Conference On*, pages 351–360. IEEE, Oct. 2013.

[167] F. Jaafar, Y.-G. Guéhéneuc, S. Hamel, F. Khomh, and M. Zulkernine. Evaluating the impact of design pattern and anti-pattern dependencies on changes and faults. *Empirical Software Engineering*, 21, Mar. 2015.

[168] S. G. James. An Interactive Interface for Refactoring Using Source Transformation. In *First International Workshop on Refactoring: Achievements, Challenges, Effects (REFACE'03)*, 2003.

[169] P. Jamshidi, M. Ghafari, A. Ahmad, and C. Pahl. A Framework for Classifying and Comparing Architecture-centric Software Evolution Research. In *2013 17th European Conference on Software Maintenance and Reengineering*, pages 305–314, Genova, Mar. 2013. IEEE.

[170] N. Japkowicz. The Class Imbalance Problem: Significance and Strategies. In *In Proceedings of the 2000 International Conference on Artificial Intelligence (ICAI*, pages 111–117, 2000.

[171] M. Jazayeri. On Architectural Stability and Evolution. In *Proceedings of the 7th Ada-Europe International Conference on Reliable Software Technologies*, Ada-Europe '02, pages 13–23, London, UK, UK, 2002. Springer-Verlag.

[172] K. Jeet and R. Dhir. Software Architecture Recovery Using Genetic Black Hole Algorithm. *SIGSOFT Softw. Eng. Notes*, 40(1):1–5, Feb. 2015.

[173] H. Kagdi. Improving change prediction with fine-grained source code mining. In *Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 559–562, Atlanta, Georgia, USA, 2007. ACM.

[174] H. Kagdi, M. L. Collard, and J. I. Maletic. Towards a taxonomy of approaches for mining of source code repositories. *MSR '05: Proceedings of the 2005 international workshop on Mining software repositories*, 30:1–5, May 2005.

[175] H. Kagdi, M. L. Collard, and J. I. Maletic. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. 2007.

[176] H. Kagdi, J. I. Maletic, and B. Sharif. Mining Software Repositories for Traceability Links. In *Proceedings of the 15th IEEE International Conference on Program Comprehension*, pages 145–154, Washington, DC, USA, 2007. IEEE Computer Society.

[177] H. Kagdi, S. Yusuf, and J. I. Maletic. Mining Sequences of Changed-files from Version Histories. In *Proceedings of the 2006 International Workshop on Mining Software Repositories*, MSR '06, pages 47–53, Shanghai, China, 2006. ACM.

[178] A. Kalenkova, W. van der Aalst, I. Lomazova, and V. Rubin. Process mining using BPMN: Relating event logs and process models. *Software & Systems Modeling*, pages 1–30, 2015.

[179] N. Karunanithi. A neural network approach for software reliability growth modeling in the presence of code churn. pages 310–317, Denver, CO, USA, 1993.

[180] A. Kaur and G. Dhiman. A Review on Search-Based Tools and Techniques to Identify Bad Code Smells in Object-Oriented Systems. In N. Yadav, A. Yadav, J. C. Bansal, K. Deep, and J. H. Kim, editors, *Harmony Search and Nature Inspired Optimization Algorithms*, volume 741, pages 909–921. Springer Singapore, Singapore, 2019.

[181] R. Kazman and Carrière. Playing Detective: Reconstructing Software Architecture from Available Evidence. *Automated Software Engineering*, 6(2):107–138, Apr. 1999.

[182] I. Keivanloo. Online Sharing and Integration of Results from Mining Software Repositories. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 1644–1646, Zurich, Switzerland, 2012. IEEE Press.

[183] P. Kelly. A congruence theorem for trees. *Pacific Journal of Mathematics*, 7(1):961–968, Mar. 1957.

[184] M. Kessentini, S. Vaucher, and H. Sahraou. Reference data. `http://www.iro.umontreal.ca/sahraouh/papers/ASE2010/`, Jan 2022.

[185] F. Khomh, S. Vaucher, Y. gaël Guéhéneuc, and H. Sahraoui. Reference data. http://www.ptidej.net/downloads/experiments/qsic09/, Jan 2022.

[186] F. Khomh, S. Vaucher, Y.-G. Guéhéneuc, and H. Sahraoui. A bayesian approach for the detection of code and design smells. In *Quality Software, 2009. QSIC'09. 9th International Conference On*, pages 305–314, 2009.

[187] C. Kiefer, A. Bernstein, and J. Tappolet. Mining Software Repositories with iSPAROL and a Software Evolution Ontology. In *Fourth International Workshop on Mining Software Repositories (MSR'07:ICSE Workshops 2007)*, pages 10–10, May 2007.

[188] M. Kim and D. Notkin. Program element matching for multi-version program analyses. In *Proceedings of the 2006 International Workshop on Mining Software Repositories - MSR '06*, page 58, Shanghai, China, 2006. ACM Press.

[189] M. Kim and D. Notkin. *Discovering and Representing Systematic Code Changes*, volume 0 of *ICSE '09*. IEEE, Los Alamitos, CA, USA, May 2009.

[190] M. Kim, D. Notkin, and D. Grossman. Automatic Inference of Structural Changes for Matching across Program Versions. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 333–343, Washington, DC, USA, 2007. IEEE Computer Society.

[191] S. Kim, K. Pan, and E. E. J. Whitehead. Memories of bug fixes. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '06/FSE-14, pages 35–45, Portland, Oregon, USA, 2006. ACM.

[192] S. Kim, K. Pan, and E. J. Whitehead. When Functions Change Their Names: Automatic Detection of Origin Relationships. In *Proceedings of the 12th Working Conference on Reverse Engineering*, WCRE '05, pages 143–152, Washington, DC, USA, 2005. IEEE Computer Society.

[193] S. Kim, T. Zimmermann, M. Kim, A. Hassan, A. Mockus, T. Girba, M. Pinzger, E. J. Whitehead, and A. Zeller. TA-RE: An Exchange

Language for Mining Software Repositories. In *Proceedings of the 2006 International Workshop on Mining Software Repositories*, MSR '06, pages 22–25, Shanghai, China, 2006. ACM.

[194] S. Kim, T. Zimmermann, K. Pan, and E. J. Jr. Whitehead. Automatic Identification of Bug-Introducing Changes. In *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*, pages 81–90, Tokyo, Sept. 2006. IEEE.

[195] S. Kim, T. Zimmermann, E. J. Whitehead, and A. Zeller. Predicting Faults from Cached History. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 489–498, Minneapolis, MN, USA, May 2007. IEEE Computer Society.

[196] S. C. Kothari, L. Bishop, J. Sauceda, and G. Daugherty. A Pattern-Based Framework for Software Anomaly Detection. *Software Quality Journal*, 12(2):99–120, June 2004.

[197] G. Lacerda, F. Petrillo, M. Pimenta, and Y. G. Guéhéneuc. Code smells and refactoring: A tertiary systematic review of challenges and observations. *Journal of Systems and Software*, 167:110610, Sept. 2020.

[198] M. Lanza. The Evolution Matrix: Recovering Software Evolution Using Software Visualization Techniques. In *Proceedings of the 4th International Workshop on Principles of Software Evolution*, IWPSE '01, pages 37–42, Vienna, Austria, 2001. ACM.

[199] L. Layman, G. Kudrjavets, and N. Nagappan. Iterative Identification of Fault-prone Binaries Using In-process Metrics. In *Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '08, pages 206–212, Kaiserslautern, Germany, 2008. ACM.

[200] C. H. Lee, C. R. Lin, and M. S. Chen. Sliding-window Filtering: An Efficient Algorithm for Incremental Mining. In *Proceedings of the Tenth International Conference on Information and Knowledge Management*, CIKM '01, pages 263–270, Atlanta, Georgia, USA, 2001. ACM.

[201] M. M. Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, 1980.

[202] M. M. Lehman and J. F. Ramil. Software Evolution and Software Evolution Processes. *Annals of Software Engineering*, 14(1):275–309, Dec. 2002.

[203] M. M. Lehman, J. F. Ramil, P. D. Wernick, D. E. Perry, and W. M. Turski. Metrics and Laws of Software Evolution - The Nineties View. In *Proceedings of the 4th International Symposium on Software Metrics*, METRICS '97, Washington, DC, USA, 1997. IEEE Computer Society.

[204] T. Lewowski and L. Madeyski. Code Smells Detection Using Artificial Intelligence Techniques: A Business-Driven Systematic Review. In *Developments in Information and Knowledge Management for Business Applications*, pages 285–319. Jan. 2022.

[205] T. Lewowski and L. Madeyski. How far are we from reproducible research on code smell detection? A systematic literature review. *Information and Software Technology*, 144:106783, Apr. 2022.

[206] H. Li and W. Cheung. An Empirical Study of Software Metrics. *IEEE Transactions on Software Engineering*, SE-13(6):697–708, June 1987.

[207] J. Li, D. Liu, and B. Yang. Process Mining: Extending α-Algorithm to Mine Duplicate Tasks in Process Logs. In K.-C. Chang, W. Wang, L. Chen, C. Ellis, C.-H. Hsu, A. Tsoi, and H. Wang, editors, *Advances in Web and Network Technologies, and Information Management*, volume 4537 of *Lecture Notes in Computer Science*, pages 396–407. Springer Berlin Heidelberg, 2007.

[208] P. L. Li, M. Shaw, J. Herbsleb, B. Ray, and P. Santhanam. Empirical Evaluation of Defect Projection Models for Widely-deployed Production Software Systems. *SIGSOFT Softw. Eng. Notes*, 29(6):263–272, Oct. 2004.

[209] Y. F. Li and H. Zhang. Integrating Software Engineering Data Using Semantic Web Technologies. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR '11, pages 211–214, Waikiki, Honolulu, HI, USA, 2011. ACM.

[210] E. Ligu, A. Chatzigeorgiou, T. Chaikalis, and N. Ygeionomakis. Identification of Refused Bequest Code Smells. In *Software Maintenance*

(ICSM), 2013 29th IEEE International Conference On, pages 392–395. IEEE, 2013.

[211] T. Lindgren and H. Boström. Resolving Rule Conflicts with Double Induction. In Berthold, H.-J. Lenz, E. Bradley, R. Kruse, and C. Borgelt, editors, *Advances in Intelligent Data Analysis V*, volume 2810 of *Lecture Notes in Computer Science*, pages 60–67. Springer Berlin Heidelberg, 2003.

[212] C. Lindig and G. Snelting. Assessing Modular Structure of Legacy Code Based on Mathematical Concept Analysis. In *Software Engineering, 1997., Proceedings of the 1997 International Conference On*, pages 349–359. IEEE, May 1997.

[213] B. Liskov. Keynote Address - Data Abstraction and Hierarchy. In *Addendum to the Proceedings on Object-oriented Programming Systems, Languages and Applications (Addendum)*, OOPSLA '87, pages 17–34, Orlando, Florida, USA, 1987. ACM.

[214] B. Livshits and T. Zimmermann. DynaMine: Finding Common Error Patterns by Mining Software Revision Histories. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, volume 30 of *ESEC/FSE-13*, pages 296–305, Lisbon, Portugal, Sept. 2005. ACM.

[215] E. M. Luks. Isomorphism of graphs of bounded valence can be tested in polynomial time. *Journal of Computer and System Sciences*, 25(1):42–65, Aug. 1982.

[216] B. Luo, Wilson, and E. R. Hancock. Spectral embedding of graphs. *Pattern Recognition*, 36(10):2213–2230, Oct. 2003.

[217] Y. Ma, T. Dey, C. Bogart, S. Amreen, M. Valiev, A. Tutko, D. Kennard, R. Zaretzki, and A. Mockus. World of code: Enabling a research workflow for mining and analyzing the universe of open source VCS data. *Empirical Software Engineering*, 26(2):22, Mar. 2021.

[218] N. R. Mabroukeh and C. I. Ezeife. A Taxonomy of Sequential Pattern Mining Algorithms. *ACM Comput. Surv.*, 43(1), Dec. 2010.

[219] J. T. Madhavan and E. J. Whitehead. Predicting buggy changes inside an integrated development environment. In *Proceedings of the 2007 OOPSLA Workshop on Eclipse Technology eXchange*, Eclipse '07, pages 36–40, Montreal, Quebec, Canada, 2007. ACM.

[220] A. Maiga, N. Ali, N. Bhattacharya, A. Sabané, Y.-G. Guéhéneuc, G. Antoniol, and E. Aimeur. Support vector machines for anti-pattern detection. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering - ASE 2012*, pages 278+, Essen, Germany, 2012. ACM Press.

[221] A. Maiga, N. Ali, N. Bhattacharya, A. Sabane, Y.-G. Guéhéneuc, G. Antoniol, and E. Aimeur. Reference data. `http://www.ptidej.net/download/experiments/ase12/`, Jan 2022.

[222] J. Maletic and M. L. Collard. Supporting source code difference analysis. In *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference On*, pages 210–219. IEEE, 2004.

[223] C. Manapragada, G. I. Webb, and M. Salehi. Extremely Fast Decision Tree. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 1953–1962, London United Kingdom, July 2018. ACM.

[224] H. Mannila and H. Toivonen. Discovering Generalized Episodes Using Minimal Occurrences. In *2nd International Conference on Knowledge Discovery and Data Mining*, pages 146–151, 1996.

[225] H. Mannila, H. Toivonen, and A. I. Verkamo. Discovery of Frequent Episodes in Event Sequences. *Data Min. Knowl. Discov.*, 1(3):259–289, Jan. 1997.

[226] M. Mäntylä and C. Lassenius. Subjective evaluation of software evolvability using code smells: An empirical study. *Empirical Software Engineering*, 11(3):395–431, Sept. 2006.

[227] O. Maqbool and H. A. Babri. Hierarchical Clustering for Software Architecture Recovery. *Transactions on Software Engineering*, 33(11):759–780, Nov. 2007.

[228] R. Marinescu. Detection Strategies: Metrics-Based Rules for Detecting Design Flaws. *Software Maintenance, IEEE International Conference on*, 0:350–359, 2004.

[229] M. Marques, J. Simmonds, P. O. Rossel, and M. C. Bastarrica. Software product line evolution: A systematic literature review. *Information and Software Technology*, 105:190–208, Jan. 2019.

[230] T. J. McCabe. A complexity measure. In *Proceedings of the 2nd International Conference on Software Engineering*, ICSE '76, pages 407+, San Francisco, California, United States, 1976. IEEE Computer Society Press.

[231] Medeiros, B. F. V. Dongen, Aalst, and A. J. M. M. Weijters. *Process Mining: Extending the Alfa-Algorithm to Mine Short Loops*. Eindhoven University of Technology, Eindhoven, 2004.

[232] A. K. Medeiros, A. J. Weijters, and W. M. Aalst. Genetic process mining: An experimental evaluation. *Data Min. Knowl. Discov.*, 14:245–304, Apr. 2007.

[233] T. Mende and R. Koschke. Revisiting the evaluation of defect prediction models. In *Proceedings of the 5th International Conference on Predictor Models in Software Engineering*, PROMISE '09, pages 1–10, Vancouver, British Columbia, Canada, 2009. ACM.

[234] J. Mendling. Event-Driven Process Chains (EPC). In *Metrics for Process Models: Empirical Foundations of Verification, Error Prediction, and Guidelines for Correctness*, pages 17–57. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[235] N. C. Mendonça and J. Kramer. Requirements for an Effective Architecture Recovery Framework. In *Joint Proceedings of the Second International Software Architecture Workshop (ISAW-2) and International Workshop on Multiple Perspectives in Software Development (Viewpoints '96) on SIGSOFT '96 Workshops*, ISAW '96, pages 101–105, San Francisco, California, USA, 1996. ACM.

[236] T. Menzies, J. Greenwald, and A. Frank. Data Mining Static Code Attributes to Learn Defect Predictors. *IEEE Transactions on Software Engineering*, 33(1):2–13, 2007.

[237] B. Michalik and D. Weyns. Architecture Query Language Framework. In *SPLC Workshops*, pages 279–286, 2010.

[238] A. Mockus and J. D. Herbsleb. Expertise browser: A quantitative approach to identifying expertise. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 503–512, Orlando, Florida, 2002. ACM.

[239] A. Mockus, N. Nagappan, and T. T. Dinh-Trong. Test coverage and post-verification defects: A multiple case study. In *Empirical Software Engineering and Measurement, 2009. ESEM 2009. 3rd International Symposium On*, ESEM '09, pages 291–301, Washington, DC, USA, Oct. 2009. IEEE.

[240] A. Mockus and L. G. Votta. Identifying reasons for software changes using historic databases. In *Software Maintenance, 2000. Proceedings. International Conference On*, pages 120–130, Victoria, BC, Canada, Aug. 2000. IEEE.

[241] N. Moha, Y.-G. Guéhéneuc, L. Duchien, and A.-F. Le Meur. DECOR: A Method for the Specification and Detection of Code and Design Smells. *Software Engineering, IEEE Transactions on*, 36(1):20–36, Jan. 2010.

[242] N. Moha, Y.-G. Gueheneuc, L. Duchien, and A.-F. L. Meur. Reference data. `http://www.ptidej.net/research/decor/`, Jan 2022.

[243] N. Moha, Y. G. Gueheneuc, A. F. Le Meur, L. Duchien, and A. Tiberghien. From a domain analysis to the specification and detection of code and design smells. *Form. Asp. Comput.*, 22:345–361, May 2010.

[244] N. Moha, Y.-g. Gueheneuc, and P. Leduc. Automatic Generation of Detection Algorithms for Design Defects. In *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*, pages 297–300, Tokyo, 2006. IEEE.

[245] R. Moser, W. Pedrycz, and G. Succi. Analysis of the reliability of a subset of change metrics for defect prediction. In *Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '08, pages 309–311, Kaiserslautern, Germany, 2008. ACM.

[246] R. Moser, W. Pedrycz, and G. Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 181–190, Leipzig, Germany, May 2008. ACM.

[247] N. Nagappan and T. Ball. Static Analysis Tools As Early Indicators of Pre-release Defect Density. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 580–586, St. Louis, MO, USA, 2005. ACM.

[248] N. Nagappan and T. Ball. Use of Relative Code Churn Measures to Predict System Defect Density. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 284–292, St. Louis, MO, USA, 2005. ACM.

[249] N. Nagappan, A. Zeller, T. Zimmermann, K. Herzig, and B. Murphy. Change Bursts As Defect Predictors. In *Proceedings of the 2010 IEEE 21st International Symposium on Software Reliability Engineering*, ISSRE '10, pages 309–318, Washington, DC, USA, 2010. IEEE Computer Society.

[250] J. Nam and S. Kim. Heterogeneous defect prediction. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 508–519, Bergamo Italy, Aug. 2015. ACM.

[251] M. Nayrolles, N. Moha, and P. Valtchev. Improving SOA antipatterns detection in Service Based Systems by mining execution traces. In *Reverse Engineering (WCRE), 2013 20th Working Conference On*, pages 321–330. IEEE, Oct. 2013.

[252] I. Neamtiu, J. S. Foster, and M. Hicks. Understanding source code evolution using abstract syntax tree matching. *SIGSOFT Softw. Eng. Notes*, 30(4):1–5, May 2005.

[253] J. Neetil and P. O. de Mendez. *Sparsity: Graphs, Structures, and Algorithms.* Springer Publishing Company, 2012.

[254] B. A. Nejmeh. NPATH: A measure of execution path complexity and its applications. *Communications of the ACM*, 31(2):188–200, Feb. 1988.

[255] S. Nelson and J. Schumann. What makes a code review trustworthy? In *System Sciences, 2004. Proceedings of the 37th Annual Hawaii International Conference On*. IEEE, Jan. 2004.

[256] E. C. Neto, D. A. da Costa, and U. Kulesza. The impact of refactoring changes on the SZZ algorithm: An empirical study. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 380–390, Mar. 2018.

[257] M. Neuhaus and H. Bunke. *Bridging the Gap Between Graph Edit Distance and Kernel Machines*. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 2007.

[258] S. Olbrich, D. S. Cruzes, V. Basili, and N. Zazworka. The evolution and impact of code smells: A case study of two open source systems. *ESEM '09: Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*, pages 390–400, 2009.

[259] S. M. Olbrich, D. S. Cruzes, and D. I. K. Sjoberg. Are all code smells harmful? A study of God Classes and Brain Classes in the evolution of three open source systems. In *Proceedings of the 2010 IEEE International Conference on Software Maintenance*, ICSM '10, Washington, DC, USA, 2010. IEEE Computer Society.

[260] Oracle. Java language specification, `https://docs.oracle.com/javase/specs/`, Nov 2016.

[261] Oracle. Java api for xml processing (jaxp), `https://docs.oracle.com/javase/tutorial/jaxp/index.html`, Nov 2021.

[262] T. J. Ostrand, E. J. Weyuker, and R. M. Bell. Programmer-based fault prediction. In *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*, PROMISE '10. ACM, 2010.

[263] J. Paakki, A. Karhinen, J. Gustafsson, L. Nenonen, and A. I. Verkamo. Software metrics by architectural pattern mining. In *In Proceedings of the International Conference on Software: Theory and Practice (16th IFIP World Computer Congress*, pages 325–332, 2000.

[264] B. Padmanabhan and A. Tuzhilin. Knowledge Refinement Based on the Discovery of Unexpected Patterns in Data Mining. *Decis. Support Syst.*, 33(3):309–321, July 2002.

[265] N. Palix, J. Lawall, and G. Muller. Tracking code patterns over multiple software versions with Herodotos. In *Proceedings of the 9th International Conference on Aspect-Oriented Software Development*, AOSD '10, pages 169–180, Rennes and Saint-Malo, France, 2010. ACM.

[266] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, D. Poshyvanyk, and A. De Lucia. Mining version histories for detecting code smells. *IEEE Transactions on Software Engineering*, 41(5):462–489, 2014.

[267] F. Palomba, G. Bavota, M. D. Penta, F. Fasano, R. Oliveto, and A. D. Lucia. Reference data. `https://dibt.unimol.it/staff/fpalomba/reports/badSmell-analysis/index.html`, Jan 2022.

[268] F. Palomba, R. Oliveto, and A. De Lucia. Investigating code smell co-occurrences using association rule learning: A replicated study. In *2017 IEEE Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE)*, pages 8–13, Feb. 2017.

[269] F. Palomba, R. Oliveto, and A. D. Lucia. Reference data. `https://dibt.unimol.it/staff/fpalomba/reports/maltesque/`, Jan 2022.

[270] C. Parnin, C. Görg, and O. Nnadi. A catalogue of lightweight visualizations to support code smell inspection. In *Proceedings of the 4th ACM Symposium on Software Visualization*, SoftVis '08, pages 77–86, Ammersee, Germany, 2008. ACM.

[271] Z. Pawlak. On learning — a rough set approach. In A. Skowron, editor, *Computation Theory*, volume 208 of *Lecture Notes in Computer Science*, pages 197–227. Springer Berlin Heidelberg, 1985.

[272] Z. Pawlak. Some Issues on Rough Sets. In J. Peters, A. Skowron, J. Grzymała-Busse, B. Kostek, R. Świniarski, and M. Szczuka, editors, *Transactions on Rough Sets I*, volume 3100 of *Lecture Notes in Computer Science*, pages 1–58. Springer Berlin Heidelberg, 2004.

[273] Ł. Puławski. Discovering Implicit Dependencies Between Source Code Artifacts by Mining Software Development Processes. In H.-D. Burkhard, P. Chrząstowski-Wachtel, L. Czaja, M. Kudlek, G. Lindemann, W. Penczek, L. Popova-Zeugmann, A. Salwicki, H. Schlingloff, A. Skowron, Z. Suraj, and M. Szczuk, editors, *Concurrency, Specification & Programming Workshop*. Uniwersystet Warszawski, Sept. 2009.

[274] Ł. Puławski. Software Defect Prediction Based on Source Code Metrics Time Series. In J. Peters, A. Skowron, C.-C. Chan, J. Grzymała-Busse, and W. Ziarko, editors, *Transactions on Rough Sets XIII*, volume 6499 of *Lecture Notes in Computer Science*, pages 104–120. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2011.

[275] Ł. Puławski. An automatic approach for detecting early indicators of design anti-patterns. In M. Virvou and S. Matsuura, editors, *JCKBSE*, volume 240 of *Frontiers in Artificial Intelligence and Applications*, pages 161–170. IOS Press, 2012.

[276] Ł. Puławski. Automatic Forecasting of Design Anti-patterns in Software Source Code. In L. Popova-Zeugmann, editor, *CS&P*, volume 928 of *CEUR Workshop Proceedings*, pages 312–323. CEUR-WS.org, 2012.

[277] Ł. Puławski. Evolutions reference data necessary to reproduce results. `https://www.mimuw.edu.pl/~lpulawski/PhD/reproduction/evolutions-dir.zip`, May 2022.

[278] Ł. Puławski. Javadoc for the this thesis source code. `https://www.mimuw.edu.pl/~lpulawski/PhD/reproduction/javadoc.zip`, May 2022.

[279] Ł. Puławski. Reference data for this thesis. `https://www.mimuw.edu.pl/~lpulawski/PhD/reproduction/expert-tagging.zip`, May 2022.

[280] Ł. Puławski. Script setenv.bat necessary to reproduce this thesis results. `https://www.mimuw.edu.pl/~lpulawski/PhD/reproduction/setEnv.bat`, May 2022.

[281] Ł. Puławski. Source code of this thesis. `https://www.mimuw.edu.pl/~lpulawski/PhD/reproduction/source.zip`, May 2022.

[282] A. Rajaraman and J. D. Ullman. *Mining of Massive Datasets*. Cambridge University Press, Dec. 2011.

[283] S. Ramanna, R. Bhatt, and P. Biernot. A Rough-Hybrid Approach to Software Defect Classification. In A. An, J. Stefanowski, S. Ramanna, C. Butz, W. Pedrycz, and G. Wang, editors, *Rough Sets, Fuzzy Sets, Data Mining and Granular Computing*, volume 4482 of *Lecture Notes in Computer Science*, pages 79–86. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2007.

[284] S. Ramanna, R. Bhatt, and P. Biernot. Software Defect Classification: A Comparative Study with Rough Hybrid Approaches. In M. Kryszkiewicz, J. Peters, H. Rybinski, and A. Skowron, editors, *Rough Sets and Intelligent Systems Paradigms*, volume 4585 of *Lecture Notes in Computer Science*, pages 630–638. Springer Berlin / Heidelberg, 2007.

[285] J. Ramil and M. Lehman. Defining and applying metrics in the context of continuing software evolution. In *Proceedings Seventh International Software Metrics Symposium*, pages 199–209, London, UK, 2000. IEEE Comput. Soc.

[286] R. Ramler, C. Klammer, and T. Natschläger. The usual suspects: A case study on delivered defects per developer. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '10, Bolzano-Bozen, Italy, 2010. ACM.

[287] D. Ratiu, S. Ducasse, T. Girba, and R. Marinescu. Using History Information to Improve Design Flaws Detection. In *Proceedings of the Eighth Euromicro Working Conference on Software Maintenance and Reengineering (CSMR'04)*, CSMR '04, Washington, DC, USA, 2004. IEEE Computer Society.

[288] J. Ratzinger, M. Fischer, and H. Gall. Improving evolvability through refactoring. *MSR '05: Proceedings of the 2005 international workshop on Mining software repositories*, 30:1–5, May 2005.

[289] J. Ratzinger, M. Pinzger, and H. Gall. EQ-Mine: Predicting Short-Term Defects for Software Evolution. In M. Dwyer and A. Lopes, editors, *Fundamental Approaches to Software Engineering*, volume 4422

of *Lecture Notes in Computer Science*, pages 12–26. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2007.

[290] J. Ratzinger, T. Sigmund, P. Vorburger, and H. Gall. Mining Software Evolution to Predict Refactoring. In *Empirical Software Engineering and Measurement, 2007. ESEM 2007. First International Symposium On*, pages 354–363, Madrid, Spain, Sept. 2007.

[291] J. L. Reutter, M. Romero, and M. Y. Vardi. Regular Queries on Graph Databases. *Theory of Computing Systems*, 61(1):31–83, July 2017.

[292] R. Robbes. Mining a Change-Based Software Repository. In *Proceedings of the Fourth International Workshop on Mining Software Repositories*, MSR '07, Washington, DC, USA, 2007. IEEE Computer Society.

[293] R. Robbes and M. Lanza. A Change-based Approach to Software Evolution. *Proceedings of the ERCIM Working Group on Software Evolution (2006)*, 166:93–109, Jan. 2007.

[294] R. Robbes, M. Lanza, and M. Lungu. An approach to software evolution based on semantic change. In *Proceedings of the 10th International Conference on Fundamental Approaches to Software Engineering*, FASE'07, pages 27–41, Braga, Portugal, 2007. Springer-Verlag.

[295] R. Robbes and G. Robles, editors. *13th International Workshop on Principles of Software Evolution, IWPSE 2013, Proceedings, August 19-20, 2013, Saint Petersburg, Russia.* ACM, 2013.

[296] M. P. Robillard and G. C. Murphy. Representing concerns in source code. *ACM Trans. Softw. Eng. Methodol.*, 16(1):3+, Feb. 2007.

[297] A. Robles-Kelly and E. R. Hancock. A Riemannian approach to graph embedding. *Pattern Recognition*, 40(3):1042–1056, Mar. 2007.

[298] G. Rodríguez-Pérez, G. Robles, A. Serebrenik, A. Zaidman, D. M. Germán, and J. M. Gonzalez-Barahona. How bugs are born: A model to identify how bugs are introduced in software components. *Empirical Software Engineering*, pages 1–47, 2020.

[299] G. Rodríguez-Pérez, A. Zaidman, A. Serebrenik, G. Robles, and J. M. González-Barahona. What if a bug has a different origin? making sense of bugs without an explicit bug introducing change. In *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '18, pages 1–4, New York, NY, USA, Oct. 2018. Association for Computing Machinery.

[300] A. Rozinat and W. M. P. van der Aalst. Decision Mining in ProM. In S. Dustdar, J. L. Fiadeiro, and A. P. Sheth, editors, *Business Process Management*, Lecture Notes in Computer Science, pages 420–425, Berlin, Heidelberg, 2006. Springer.

[301] J. R. Ruthruff, J. Penix, J. D. Morgenthaler, S. Elbaum, and G. Rothermel. Predicting accurate and actionable static analysis warnings: An experimental approach. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 341–350, Leipzig, Germany, 2008. ACM.

[302] F. Sabir, F. Palma, G. Rasool, Y.-G. Guéhéneuc, and N. Moha. A systematic literature review on the detection of smells and their evolution in object-oriented and service-oriented systems. *Software: Practice and Experience*, 49(1):3–39, Jan. 2019.

[303] S. L. Salzberg. C4.5: Programs for Machine Learning by J. Ross Quinlan. Morgan Kaufmann Publishers, Inc., 1993. *Machine Learning*, 16(3):235–240, Sept. 1994.

[304] K. Sartipi. Software architecture recovery based on pattern matching. In *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference On*, pages 293–296. IEEE, 2003.

[305] W. Scacchi. Understanding Open Source Software Evolution. In *Applying, Breaking, and Rethinking the Laws of Software Evolution.* John Wiley & Sons, 2003.

[306] J. G. Schneider, R. Vasa, and L. Hoon. Do metrics help to identify refactoring? In *Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE)*, IWPSE-EVOL '10, pages 3–7, Antwerp, Belgium, 2010. ACM.

[307] N. Seliya and T. Khoshgoftaar. Software quality estimation with limited fault data: A semi-supervised learning perspective. *Software Quality Journal*, 15(3):327–344, Sept. 2007.

[308] N. Shi and R. A. Olsson. Reverse Engineering of Design Patterns from Java Source Code. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, ASE '06, pages 123–134, Washington, DC, USA, 2006. IEEE Computer Society.

[309] E. Shihab, Z. M. Jiang, W. M. Ibrahim, B. Adams, and A. E. Hassan. Understanding the impact of code and process metrics on post-release defects: A case study on the Eclipse project. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '10, pages 1–10, Bolzano-Bozen, Italy, 2010. ACM.

[310] M. Siff and T. Reps. Identifying Modules via Concept Analysis. *IEEE Trans. Softw. Eng.*, 25(6):749–768, Nov. 1999.

[311] J. Śliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? In *MSR '05: Proceedings of the 2005 International Workshop on Mining Software Repositories*, pages 1–5, St. Louis, Missouri, 2005. ACM.

[312] M. Sokolova and G. Lapalme. A systematic analysis of performance measures for classification tasks. *Information Processing & Management*, 45(4):427–437, July 2009.

[313] Q. Song, M. Shepperd, M. Cartwright, and C. Mair. Software Defect Association Mining and Defect Correction Effort Prediction. *IEEE Transactions on Software Engineering*, 32(2):69–82, 2006.

[314] S. Sorlin and C. Solnon. Reactive Tabu Search for Measuring Graph Similarity. In *In the 5th IAPR Workshop on Graph-based Representations in Pattern Recognition (GbR 2005), LNCS 3434*, pages 172–182, 2005.

[315] K. Stencel and P. Wegrzynowicz. Detection of Diverse Design Pattern Variants. In *2008 15th Asia-Pacific Software Engineering Conference*, pages 25–32, Beijing, China, Dec. 2008. IEEE.

[316] A. Stoianov and I. Sora. Detecting patterns and antipatterns in software using Prolog rules. In *2010 International Joint Conference on Computational Cybernetics and Technical Informatics*, pages 253–258, Timisoara, May 2010. IEEE.

[317] C. Sun, H. Zhang, J. G. Lou, H. Zhang, Q. Wang, D. Zhang, and S. C. Khoo. Querying Sequential Software Engineering Data. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 700–710, Hong Kong, China, 2014. ACM.

[318] P. Synak, J. G. Bazan, A. Skowron, and J. F. Peters. Spatio-Temporal Approximate Reasoning over Complex Objects. *Fundam. Inf.*, 67(1-3):249–269, Feb. 2005.

[319] D. H. Taenzer, M. Ganti, and S. Podar. Problems in Object-Oriented Software Reuse. In S. Cook, editor, *ECOOP '89: Proceedings of the Third European Conference on Object-Oriented Programming, Nottingham, UK, July 10-14, 1989*, pages 25–38. Cambridge University Press, 1989.

[320] M. K. Thota, F. H. Shajin, and P. Rajesh. Survey on software defect prediction techniques. *International Journal of Applied Science and Engineering*, 17(4):331–344, Dec. 2020.

[321] Tomsk State University, O. A. Zmeev, and L. S. Ivanova. Design artifacts detection. Review of the approaches. *Vestnik Tomskogo gosudarstvennogo universiteta. Upravlenie, vychislitel'naya tekhnika i informatika*, (2(31)):81–90, June 2015.

[322] J. Torres Carbonell and J. Parets-Llorca. A Formalisation of the Evolution of Software Systems. In P. Kopacek, R. Moreno-Díaz, and F. Pichler, editors, *Computer Aided Systems Theory - EUROCAST'99*, volume 1798 of *Lecture Notes in Computer Science*, pages 435–449. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2000.

[323] R. P. F. Trindade, M. A. da Silva Bigonha, and K. A. M. Ferreira. Oracles of Bad Smells: A Systematic Literature Review. In *Proceedings of the 34th Brazilian Symposium on Software Engineering*, pages 62–71, Natal Brazil, Oct. 2020. ACM.

[324] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S. T. Halkidis. Design Pattern Detection Using Similarity Scoring. *Software Engineering, IEEE Transactions on*, 32(11):896–909, Nov. 2006.

[325] R. S. Tsay. *Analysis of Financial Time Series*. Wiley, third edition, Aug. 2010.

[326] Q. Tu and M. W. Godfrey. An Integrated Approach for Studying Architectural Evolution. In *Proceedings of the 10th International Workshop on Program Comprehension*, IWPC '02, Washington, DC, USA, 2002. IEEE Computer Society.

[327] W. van der Aalst, T. Weijters, and L. Maruster. Workflow Mining: Discovering Process Models from Event Logs. *Knowledge and Data Engineering, IEEE Transactions on*, 16(9):1128–1142, Sept. 2004.

[328] W. M. P. van der Aalst. Formalization and verification of event-driven process chains. *Information and Software Technology*, 41(10):639–650, July 1999.

[329] W. M. P. van der Aalst and A. H. M. T. Hofstede. YAWL: Yet Another Workflow Language. *Inf. Syst.*, 30(4):245–275, June 2005.

[330] W. M. P. van der Aalst and B. F. Van Dongen. Discovering Workflow Performance Models from Timed Logs. In *International Conference on Engineering and Deployment of Cooperative Information Systems (EDCIS 2002), Volume 2480 of Lecture Notes in Computer Science*, volume 2480, pages 45–63, 2002.

[331] A. van Deursen and T. Kuipers. Identifying Objects Using Cluster and Concept Analysis. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, pages 246–255, Los Angeles, California, USA, 1999. ACM.

[332] S. Vaucher and H. Sahraoui. Do software libraries evolve differently than applications?: An empirical investigation. In *Proceedings of the 2007 Symposium on Library-Centric Software Design*, LCSD '07, pages 88–96, Montreal, Canada, 2007. ACM.

[333] L. Voinea and A. Telea. Multiscale and multivariate visualizations of software evolution. In *Proceedings of the 2006 ACM Symposium on*

Software Visualization, SoftVis ’06, pages 115–124, Brighton, United Kingdom, 2006. ACM.

[334] L. Voinea and A. Telea. Visual querying and analysis of large software repositories. *Empirical Software Engineering*, 14(3):316–340, June 2009.

[335] M. von Detten and M. Platenius-Mohr. Improving Dynamic Design Pattern Detection in Reclipse with Set Objects. In *7th International Fujaba Days*, pages 15–19, Jan. 2009.

[336] B. Walter and B. Pietrzak. Multi-criteria Detection of Bad Smells in Code with UTA Method. In H. Baumeister, M. Marchesi, and M. Holcombe, editors, *Extreme Programming and Agile Processes in Software Engineering*, volume 3556 of *Lecture Notes in Computer Science*, pages 1159–1161. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2005.

[337] M. Wedel, U. Jensen, and P. Göhner. Mining software code repositories and bug databases using survival analysis models. In *Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM ’08, pages 282–284, Kaiserslautern, Germany, 2008. ACM.

[338] P. Wegrzynowicz and K. Stencel. Relaxing queries to detect variants of design patterns. In *Computer Science and Information Systems (FedCSIS), 2013 Federated Conference On*, pages 1571–1578. IEEE, 2013.

[339] P. Weissgerber and S. Diehl. Identifying Refactorings from Source-Code Changes. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, pages 231–240, Washington, DC, USA, 2006. IEEE Computer Society.

[340] L. Wen, W. M. Aalst, J. Wang, and J. Sun. Mining Process Models with Non-free-choice Constructs. *Data Min. Knowl. Discov.*, 15(2):145–180, Oct. 2007.

[341] G. Widmer and M. Kubat. Learning in the Presence of Concept Drift and Hidden Contexts. *Mach. Learn.*, 23(1):69–101, Apr. 1996.

[342] R. Wieman. *Anti-Pattern Scanner: An Approach to Detect Anti-Patterns and Design Violations.* LAP LAMBERT Academic Publishing, Nov. 2011.

[343] William. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis.* John Wiley & Sons, first edition, Apr. 1998.

[344] M. Wojnarski. Nondeterministic Discretization of Weights Improves Accuracy of Neural Networks. In J. Kok, J. Koronacki, R. Mantaras, S. Matwin, D. Mladenič, and A. Skowron, editors, *Machine Learning: ECML 2007*, volume 4701 of *Lecture Notes in Computer Science*, pages 765–772. Springer Berlin Heidelberg, 2007.

[345] S. K. M. Wong, W. Ziarko, and U. of Regina. Department of Computer Science. *On Optimal Decision Rules in Decision Tables.* Technical Report (University of Regina. Department of Computer Science). University of Regina, Computer Science Department, 1985.

[346] J. Wu, R. C. Holt, and A. E. Hassan. Exploring Software Evolution Using Spectrographs. In *Proceedings of the 11th Working Conference on Reverse Engineering*, pages 80–89, Washington, DC, USA, 2004. IEEE Computer Society.

[347] T.-K. Wu, S.-C. Huang, Y.-R. Meng, and Y.-C. Lin. Improving Rules Quality Generated by Rough Set Theory for the Diagnosis of Students with LDs through Mixed Samples Clustering. In P. Wen, Y. Li, L. Polkowski, Y. Yao, S. Tsumoto, and G. Wang, editors, *Rough Sets and Knowledge Technology*, volume 5589 of *Lecture Notes in Computer Science*, pages 94–101. Springer Berlin Heidelberg, 2009.

[348] Y. Wu, R. A. Gandhi, and H. Siy. Using Semantic Templates to Study Vulnerabilities Recorded in Large Software Repositories. In *Proceedings of the 2010 ICSE Workshop on Software Engineering for Secure Systems*, SESS '10, pages 22–28, Cape Town, South Africa, 2010. ACM.

[349] Z. Xing and E. Stroulia. Analyzing the Evolutionary History of the Logical Design of Object-Oriented Software. *IEEE Trans. Softw. Eng.*, 31:850–868, Oct. 2005.

[350] Z. Xing and E. Stroulia. UMLDiff: An algorithm for object-oriented design differencing. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ASE '05, pages 54–65, Long Beach, CA, USA, 2005. ACM.

[351] A. Yamashita and L. Moonen. Exploring the impact of inter-smell relations on software maintainability: An empirical study. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 682–691, San Francisco, CA, USA, May 2013. IEEE.

[352] K. Yamashita, C. Huang, M. Nagappan, Y. Kamei, A. Mockus, A. E. Hassan, and N. Ubayashi. Thresholds for Size and Complexity Metrics: A Case Study from the Perspective of Defect Density. In *2016 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 191–201, Aug. 2016.

[353] X. Yan, P. S. Yu, and J. Han. Graph Indexing: A Frequent Structure-based Approach. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, pages 335–346, Paris, France, 2004. ACM.

[354] G.-E. Yap, X.-L. Li, and P. Yu. Effective Next-Items Recommendation via Personalized Sequential Pattern Mining. In S.-g. Lee, Z. Peng, X. Zhou, Y.-S. Moon, R. Unland, and J. Yoo, editors, *Database Systems for Advanced Applications*, volume 7239 of *Lecture Notes in Computer Science*, pages 48–64. Springer Berlin Heidelberg, 2012.

[355] X. Ying. An Overview of Overfitting and its Solutions. *J. Phys.: Conf. Ser.*, 1168, Feb. 2019.

[356] D. Yu, P. Zhang, J. Yang, Z. Chen, C. Liu, and J. Chen. Efficiently detecting structural design pattern instances based on ordered sequences. *Journal of Systems and Software*, 142:35–56, Aug. 2018.

[357] T. Yu, S. Simoff, and T. Jan. VQSVM: A case study for incorporating prior domain knowledge into inductive machine learning. *Neurocomputing*, 73(13-15):2614–2623, Aug. 2010.

[358] L. A. Zadeh. Toward a theory of fuzzy information granulation and its centrality in human reasoning and fuzzy logic. *Fuzzy Sets and Systems*, 90(2):111–127, Sept. 1997.

[359] L. A. Zadeh. From Computing with Numbers to Computing with Words - From Manipulation of Measurements to Manipulation of Perceptions. In *Intelligent Systems and Soft Computing: Prospects, Tools and Applications*, pages 3–40, London, UK, UK, 2000. Springer-Verlag.

[360] M. Zelkowitz, S. Forrest, B. Vic, W. Brown, P. Costa, M. Lindvall, D. Port, I. Rus, and R. Tesoriero. What we have learned about fighting defects. In *8th IEEE Symposium on Software Metrics*, Nov. 2002.

[361] M. Zhang, N. Baddoo, P. Wernick, and T. Hall. Improving the Precision of Fowler's Definitions of Bad Smells. In *2008 32nd Annual IEEE Software Engineering Workshop*, pages 161–166, Kassandra, Greece, Oct. 2008. IEEE.

[362] X. Zhang, M. Luo, and D. Pi. Effective Classifier Pruning with Rule Information. In A. Hoffmann, H. Motoda, and T. Scheffer, editors, *Discovery Science*, volume 3735 of *Lecture Notes in Computer Science*, pages 392–395. Springer Berlin Heidelberg, 2005.

[363] P. Zhao, J. X. Yu, and P. S. Yu. Graph Indexing: Tree + Delta <= Graph. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, pages 938–949, Vienna, Austria, 2007. VLDB Endowment.

[364] L. Zhu, M. A. Babar, and R. Jeffery. Mining patterns to support software architecture evaluation. In *Software Architecture, 2004. WICSA 2004. Proceedings. Fourth Working IEEE/IFIP Conference On*, pages 25–34. IEEE, June 2004.

[365] T. Zimmermann. Changes and bugs Mining and predicting development activities. In *Software Maintenance, 2009. ICSM 2009. IEEE International Conference On*, pages 443–446. IEEE, 2009.

[366] T. Zimmermann, S. Diehl, and A. Zeller. How History Justifies System Architecture (or Not). In *Proceedings of the 6th International Workshop on Principles of Software Evolution*, Washington, DC, USA, 2003. IEEE Computer Society.

[367] T. Zimmermann and N. Nagappan. Predicting defects using network analysis on dependency graphs. In *Proceedings of the 30th International*

Conference on Software Engineering, ICSE '08, pages 531–540, Leipzig, Germany, 2008. ACM.

[368] T. Zimmermann and N. Nagappan. Predicting defects with program dependencies. In *ESEM '09: Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*, pages 435–438, Washington, DC, USA, 2009. IEEE Computer Society.

[369] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, and B. Murphy. Cross-project Defect Prediction: A Large Scale Experiment on Data vs. Domain vs. Process. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, pages 91–100, Amsterdam, The Netherlands, 2009. ACM.

[370] T. Zimmermann, R. Premraj, and A. Zeller. Predicting Defects for Eclipse. In *Predictor Models in Software Engineering, 2007. PROMISE07: ICSE Workshops 2007. International Workshop On*, volume 0 of *PROMISE '07*, page 9, Minneapolis, MN, USA, May 2007. IEEE.

[371] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl. Mining version histories to guide software changes. *Software Engineering, IEEE Transactions on*, 31(6):429–445, June 2005.

Appendix A

Detailed description of experiments

This chapter presents the results of experiments conducted within this research and referenced in the preceding chapters. Each section provides specific information about ways to reproduce a respective experiment. General instructions on the reproduction of all experiments is given in Appendix B.

A.1 Software snapshot model simplifications

A.1.1 Equinominal methods

Table A.1 shows how many methods were overloaded in the source code of the analyzed system. The cell in the column labeled *Name* presents the fractions of overloaded methods relative to the total number of all methods declared in a given source code. The cell in the column labeled *Name + arguments* presents similarly defined fractions of overloaded methods such that at least one of the overloading methods has the same number of arguments. The cell in the column labeled *Name + arguments + return type* presents similarly defined fractions of overloaded methods such that at least one of the overloading methods has the same number of arguments and a different return type. As the model proposed in this thesis equates overloaded methods with the same number of arguments, the values in the last two columns show what fraction of the actual source code is wrongly represented in the model and how it affects the evaluation of the call graph, respectively. The error

Dataset	Name	Name + arguments	Name + arguments + return type
JHotDraw	3,06%	0,20%	0,00%
ArgoUML	1,58%	0,17%	0,02%
Wildfly	1,60%	0,18%	0,08%
Elasticsearch	4,23%	0,73%	0,57%
Lucene solr	2,34%	0,26%	0,11%
Xerces	2,61%	0,32%	0,08%
Struts	2,73%	0,10%	0,04%

Table A.1: The fraction of equinominal methods in experimentally used data.

appears to be insignificant. To reproduce this experiment run class UninomialMethodsCountExperient.

A.2 Locality properties

Experiments in this section validate *locality properties* described in Section 6.2.3.

A.2.1 Size of a typical commit

Table A.2 shows the distribution of the number of files which are modified within a single commit. The values in the cells give information what fraction of all commits satisfy the condition: "the number of files modified in the commit was within the range specified in the column header".

The table shows that in the course of development of the analyzed systems, the number of files modified by a commit is typically relatively low, depending on the system: not exceeding 5 files in the case of 64-96%, and not exceeding 10 files in the case of 76-99% commits. The total number of source code files in these systems ranges from thousands to tens of thousands, making the relative size of a change very insignificant. To reproduce this experiment run class LocalityPropertiesExperiment.

Size of commit	[0-5]	(5-10]	(10-25]	(25-50]	(50-100]
JHotDraw	64%	12%	11%	8%	5%
ArgoUML	96%	3%	1%	0%	0%
Wildfly	70%	12%	10%	5%	3%
Elasticsearch	79%	9%	8%	4%	1%
Lucene solr	90%	4%	4%	1%	1%
Xerces	90%	5%	3%	1%	1%
Struts	86%	6%	5%	2%	1%

Table A.2: The approximate distribution of the number of source code files modified within a single commit.

A.2.2 Size of $Files_?^{aff}$ set

The practical savings from the use of adaptive algorithm described in section 6.2.3, are directly related to the fraction of files that are *affected*[1] by a single commit. To be more precise, we can say that it is the fraction of all such files relative to the number of all files in the program source code.

This experiment verifies the fraction of files affected by a single commit. Technically, it analyzes the number of files in the $Files_{(r_1,r_2)}^{aff}$ relative to the number of files $Files_{r_1}$, where r_1 and r_2 are consecutive revisions. Please recall from section 6.2.3 that the adaptive construction of system evolution is motivated by the fact that only files from $Files_?^{aff}$ need to be parsed at each revision. Thus, the smaller this set, the more efficient the adaptive algorithm. The results of this experiments are presented in table A.3: Cells in this table give information on the fraction of commits that satisfy the following condition: "the ratio of $\frac{|Files_{(r_1,r_2)}^{aff}|}{|Files_{r_1}|}$ satisfies the condition from the column header, where the revision of the commit is r_2 and the preceding revision in software evolution is r_1."

For example, the value 57% in the cell in row *JHotDraw* and column $\leqslant 5\%$ indicates that in 86% of all commits in the analyzed evolution of JHotDraw system, the power of $Files_?^{aff}$ was within 0 and 5% of the power of $Files$ right before the commit". Other cells in the table can be read accordingly.

To reproduce the experiment, run class AffectedClassesSizeExperiment.

[1]The definition of *affected file* is given in 6.2.3.

Dataset	$\leqslant 5\%$	$\leqslant 10\%$	$\leqslant 25\%$	$\leqslant 50\%$
JHotDraw	57%	70%	83%	91%
ArgoUML	87%	93%	97%	99%
Wildfly	72%	85%	95%	98%
Elasticsearch	79%	88%	95%	98%
Lucene solr	81%	89%	96%	99%
Struts	99%	99%	99%	99%
Xerces	86%	94%	98%	99%

Table A.3: The approximate distribution of the power of $Files^{aff}$ relative to the number of files in the analyzed system.

A.2.3 Volume of a fragment of a system under development

The preceding experiments refer to the volume of modification in the system source code that are performed within a single commit. Yet, there still might be a situation when many frequent commits altogether modify a large fragment of the system source code, while each separately modifies only a small portion. Naturally, in such case the benefits arising from the adaptive evaluation of the software evolution are reduced.

This experiment shows what fraction of the system source code is typically under development. For each commit c that took place at time t and period length p (e.g. $p = 1\text{hour}$) we can calculate how many files were modified in c and all other commits that preceded c and took place after time $t - p$. Selected results are presented in Table A.4. Each column defines a condition on the number of modified files. Each row is associated with a certain fixed period. Each cell gives the fraction of commits which satisfy the condition indicated by the column in the period designated by the row.

For example the value 67% in the first cell shows that 67% of commits in the analyzed evolution of JHotDraw satisfied the condition: "the number of files modified in the commit and later than $1h$ before it did not exceed 50". Clearly, large values of data presented in the table implicitly indicate the efficiency of an adaptive algorithm described in section 6.2.3, as they mean that at the same time only a small fraction of the system source code is being modified. This conceptual characteristic is called *temporal locality.* Any analyzed system usually bears this characteristic.

An interesting observation that can be drawn from this experiment is that evolutions of some systems, namely: Wildfly and Elasticsearch, tend to lack temporal locality. One can hypothesize that this is due to the fact that both systems are popular, open-source, community-developed initiatives with large numbers of active committers (around 350 and 1700, respectively). To reproduce the experiment run class LocalityPropertiesExperiment.

A.3 Pattern instances changing over time

This experiment validates statements from Section 6.5.6, by showing how often a pattern instance changes the set of its entities between subsequent revisions.

We say that a pattern instance is *entity-modified* by revision r if it is present at revision r and at the revision that directly precedes r and the set of entities of this pattern entity differs between these two revisions. (see Section 6.3.1 for formal definitions). Let pair (i, r) denote the fact that a pattern instance with identity i was present at revision r in the analyzed system. Table A.5 shows the fraction of such pairs where the pattern instance with identity i was entity-modified by revision r. To reproduce this experiment, run class ChangeOfPatternInstanceEntitiesSizeExperiment.

A.4 D-bounded classifiers

This section describes experiments related to the concept of *D-bounded classifier* defined in Section 6.3.4.

A.4.1 The fraction of software entities that are D-reachable from affected entity

Table A.6 shows the approximate distribution of the fraction of entities that are D-reachable from an affected entity, where D corresponds to the respective classifier for design anti-pattern defined in Section 6.4. Usually, in all considered types of a design anti-pattern in at least half of all revisions the fraction of such entities did not exceed 1%. To reproduce this experiment run class SizeOfGraphBoundedByPathStartingAtAffectedEntitiesExperiment.

Number of files	$\leqslant 50$	$\leqslant 100$	$\leqslant 250$	$\leqslant 500$
JHotDraw				
1 hour	67%	83%	94%	96%
2 hours	67%	82%	93%	96%
1 day	59%	77%	92%	96%
3 days	50%	73%	91%	96%
Wildfly				
1 hour	4%	7%	14%	25%
2 hours	4%	7%	14%	25%
1 day	4%	7%	14%	25%
3 days	3%	6%	12%	22%
ArgoUML				
1 hour	82%	91%	97%	99%
2 hours	80%	90%	96%	97%
1 day	67%	85%	95%	96%
3 days	45%	70%	93%	96%
Elasticsearch				
1 hour	13%	15%	20%	26%
2 hours	12%	15%	20%	26%
1 day	10%	14%	19%	26%
3 days	10%	14%	19%	26%
Lucene solr				
1 hour	74%	84%	94%	97%
2 hours	72%	84%	93%	97%
1 day	51%	72%	91%	97%
3 days	23%	48%	84%	97%
Xerces				
1 hour	75%	85%	94%	98%
2 hours	71%	84%	93%	98%
1 day	69%	79%	91%	97%
3 days	65%	72%	84%	97%
Struts				
1 hour	72%	77%	80%	88%
2 hours	81%	77%	80%	83%
1 day	86%	73%	98%	99%
3 days	86%	61%	98%	99%

Table A.4: The fraction of commits that modify the number of files given in column headers.

Dataset	Fraction of entity-modified pattern instances
Elasticsearch	2 ‰
JHotDraw	5 ‰
Lucene solr	1 ‰
Struts	2 ‰
Wildfly	6 ‰
Others	0 ‰

Table A.5: The fraction of entity-modified pattern instances in the entire evolution of the analyzed systems

A.4.2 Fraction of entities that are 1-distance-close to the instance of some anti-pattern

This experiment empirically validates the efficiency of adaptive heuristics described in Section 6.6 and 6.5.6. The values given in table A.7, show that, typically, in as many as 99% of cases only less than 5% of software entities are 1-distance-close to a pattern instance. This statement does not hold for Struts, for which we can state that always the number of entities 1-distance-close to a pattern does not exceed 10% and 25% of the number of all entities respectively. To reproduce this experiment run class StaticPatternsNeighborhoodSizeExperiment.

A.4.3 Longest containment, call and inheritance paths

Efficient evaluation of some metrics (see Section 6.2.1) and complexity of some classifiers of design anti-patterns (see Section 6.4) rely on three numbers:

- H - the maximum depth of inheritance tree, which is equal to the length of the longest path in SSn built from edges labeled *extend*,
- T - the maximum depth of containment relation, which is equal to the length of the longest path in SSn built from edges labeled *contain* and
- C - the maximum length of call path. Here a *call path* is every path that: connects any two nodes e_1 and e_2, is built from edges labeled *call* only and there is no shorter path built only from *call* edges that connects e_1 and e_2.

Dataset	Paths for pattern	$\leqslant 1\%$	$\leqslant 2\%$	$\leqslant 5\%$	$\leqslant 10\%$
argouml	SAK	82%	83%	83%	86%
	YOYO	89%	89%	91%	93%
	BB	83%	83%	84%	86%
	ANEMIC	83%	84%	89%	92%
	BC/Blob	83%	83%	86%	90%
lucene	SAK	73%	73%	73%	73%
	YOYO	83%	83%	88%	89%
	BB	73%	73%	74%	75%
	ANEMIC	73%	73%	75%	76%
	GC/BC/Blob	73%	73%	74%	74%
elasticsearch	SAK	53%	54%	56%	58%
	YOYO	58%	60%	66%	68%
	BB	54%	55%	57%	59%
	ANEMIC	56%	59%	63%	74%
	BC/Blob	55%	57%	61%	68%
jhotdraw	SAK	44%	44%	46%	69%
	YOYO	60%	62%	71%	74%
	BB	44%	45%	51%	57%
	ANEMIC	44%	46%	52%	59%
	BC/Blob	44%	45%	48%	56%
struts	SAK	68%	68%	71%	71%
	YOYO	69%	70%	70%	72%
	BB	69%	71%	73%	76%
	ANEMIC	71%	74%	78%	81%
	BC/Blob	71%	73%	76%	80%
wildfly	SAK	39%	39%	42%	45%
	YOYO	53%	55%	58%	60%
	BB	40%	45%	50%	56%
	ANEMIC	48%	52%	56%	59%
	BC/Blob	42%	47%	54%	59%
xerces	SAK	69%	71%	71%	73%
	YOYO	72%	72%	74%	74%
	BB	70%	71%	73%	73%
	ANEMIC	70%	71%	73%	75%
	BC/Blob	70%	71%	73%	75%

Table A.6: The approximate distribution of the number of entities that are D-reachable from affected entities. The values in the cells show the fraction of all revisions in system evolution in which the number of all entities that are D-reachable from an *affected entity* (see Definition 21) does not exceed the threshold given in the cell. D in the above definition corresponds to the paths related to the classifier of a specific design anti-pattern, as defined in Section 6.4, where: SAK=Swiss Army Knife, BC=Brain Class, BB=Base Bean, ANEMIC=Anemic Entity. For example: the value 82% in segment related to argouml, column "$\leqslant 1\%$" and row SAK, means that in 82% of revisions in the evolution of argouml software (see Section 6.7.1) the number of entities that were $D_H -$ reachable from an entity affected did not exceed 1% of all entities, where D_H is defined as in Fact 6 (the Fact states that the classifier for Swiss Army Knife is D_H-bounded).

Dataset	$\leqslant 1\%$	$\leqslant 2\%$	$\leqslant 5\%$	$\leqslant 10\%$	$\leqslant 25\%$
JHotDraw	91%	99%	100%	100%	100%
ArgoUML	62%	84%	100%	100%	100%
Wildfly	100%	100%	100%	100%	100%
Lucene solr	90%	94%	99%	100%	100%
elasticsearch	75%	98%	99%	100%	100%
Struts	72%	90%	99%	100%	100%
Xerces	11%	29%	42%	44 %	100%

Table A.7: The approximate distribution of the fraction of entities (relative to the number of all software entities) which are at most 1-close to a pattern instance during the entire software evolution. The values in the cells correspond to the fraction of all occurrences of pattern instances in which the fraction of entities 1-distance-close to the instance of a pattern indicated by the row did not exceed the threshold indicated in the column.

Tables A.8, A.9 and A.10 show respectively: the distribution of inheritance depth, the distribution of containment depth and the distribution of call path length in the entire evolution of systems analyzed in this research.

Inheritance depth does not exceed 7 and classes with $DIT > 4$ tend to be very infrequent. The depth of containment relation does not exceed 5 and 3 is by far the most frequent value. The maximum length of a call path does not exceed 12 and such paths longer than 6 are very infrequent.

To reproduce the computation of the these values run class HiearchyDepthDitributionExperiment, class ContainmentDepthDistributionExperiment and class CallDepthDistributionExperiment respectively.

A.5 Static pattern detection

A.5.1 Quality of detection of exemplary patterns

Table A.11 shows the quality of classification of selected anti-patterns and code smells according to definable patterns described in Section 6.4. These models were tested on datasets described in Section 6.7.1 and experimentally validated against expert tagging described in Appendix B.1. To reproduce this result run class FindStaticPatternsExperiment with property INCLUDE_SPATIO_TEMPORAL_RULES=false.

Dataset	0	1	2	3	4	5	6	7	8
JHotDraw	77%	18%	4%	2%	0%	-	-	-	-
ArgoUML	44%	34%	15%	5%	1%	0%	-	-	-
Xerces	55%	21%	12%	7%	4%	2%	-	-	-
Wildfly	80%	14%	4%	2%	0%	0%	0%	-	-
Elasticsearch	50%	19%	18%	7%	4%	2%	0%	0%	-
Lucene solr	40%	30%	19%	8%	3%	0%	-	-	-
Struts	45%	31%	14%	7%	2%	1%	-	-	-

Table A.8: The approximate distribution of the depth of inheritance tree in the entire evolution of the analyzed systems. Each cell provides information on the fraction of classes from the source code of the system represented by the row which have the depth of inheritance tree represented by the column. "-" in the cell means that there are no classes with a respective depth. Please note that this depth is measured only within the source code of the system. Therefore, a class which inherits from java.lang.Object has depth = 0.

Dataset	2	3	4	5
JHotDraw	13%	87%	0%	-
ArgoUML	13%	87%	0%	-
Xerces	9%	91%	0%	-
Wildfly	19%	81%	0%	-
Elasticsearch	14%	86%	0%	-
Lucene solr	16%	84%	0%	-
Struts	10%	90%	0%	-

Table A.9: The approximate distribution of the depth of containment in the entire evolution of the analyzed systems. Each cell provides information on the fraction of entities from the source code of the system represented by the row that have containment depth represented by the column. "-" in the cell means that there are no entities with a respective depth.

Dataset	1	2	3	4	5	6	7	8	9	10	11	12
JHotD.	100	98	80	36	8	1	0	0	-	-	-	-
Argo.	100	99	77	28	6	1	0	0	-	-	-	-
Xerces	99	97	77	34	7	0	0	0	0	-	-	-
Wildfly	100	100	93	64	29	9	2	0	0	0	0	0
Elastic.	100	100	83	37	10	2	0	0	0	0	-	-
Lucene s.	100	99	79	26	6	1	0	0	-	-	-	-
Struts	99	96	79	45	17	6	1	0	-	-	-	-

Table A.10: The approximate distribution of the length of call paths in the entire evolution of the analyzed systems. Each cell provides information about the percent of methods from the source code of the system represented by the row that have an outgoing call path of a length not smaller than the value in the column header. For example 98 in the cell "JHotDraw\2" means that 98% of all methods from JHotDraw was a starting node of a call path whose length was at last 2. "-" in the cell means that there were no respective methods.

A.5.2 Index-based search

Table A.12 shows the experimental validation of the concept of metrics filter described in Section 6.3.3. Each row represents a type of software pattern and a simple metrics-based filter, which corresponds to the definition of respective classifier described in Section 6.4. They are correlated in such a way that each instance of the pattern must contain a node representing the code entity, whose metrics satisfy the condition given in the second column. For example, each instance of the *YoYo* pattern must contain a class whose *depth of inheritance tree* exceeds 5. The cells with numeric data show what fraction of software entities satisfy the metrics filter condition in the respective dataset. To reproduce this result run class MetricsSelectivityExperiment.

anti-pattern type / test data	Swiss Army Knife	Blob	Data clumps	Base bean	Brain Class	YoYo	Anemic entity
Argo Uml	0.78/1.0	0.90/0.76	N/A	0.71/0.88	N/A	1.0/1.0	1.0/1.0
Elasticsearch	0.78/0.99	0.83/0.9	0.99/0.96	0.71/0.88	0.87/0.84	0.98/1.0	1.0/1.0
JHotDraw	1.0 /0.91	1.0/1.0	0.28/0.0	N/A	0.0/0.0	N/A	N/A
Lucene	0.86/1.0	0.88/0.9	N/A	0.97/1.0	0.95/0.78	1.0/1.0	N/A
Struts	0.99/1.0	N/A	0.98/0.1	N/A	N/A	N/A	N/A
Wildfly	0.94/1.0	0.92/1.0	0.99/1.0	1.0/1.0	N/A	N/A	1.0/1.0
Xerces	0.8/0.89	0.91/0.69	0.99/0.84	N/A	N/A	0.98/1.0	N/A

Table A.11: The table shows the quality of detection of instances of the design anti-patterns defined in columns in the source code defined by the row. The static detection methods described in Section 6.4 were used without spatio-temporal enhancement described in Section 6.7.4. Each cell contains two numbers: precision/recall. N/A in the cell indicates that there were no instances.

A.6 Quality of prediction of spatio-temporal rules

A.6.1 Detecting occurrences of design patterns

The following Tables A.13 - A.22 present the quality of the method for predicting occurrences of pattern instances by spatio-temporal rules. This is the experimental validation of the key contribution of the present thesis. Each row corresponds to the system whose evolution was used to train the classifier according to the description in Section 6.6 with 1-distance-closeness and 2-distance remoteness as a basic spatial relations. Each column corresponds to a system whose evolution was used to verify the quality of prediction of the rules. Each cell contains a respective pair: (*precision*/*recall*) which describes the quality of prediction of the rules trained on the evolution of the system from the row on the evolution of the system from the column. These measures are computed according to the description in Section 6.7.3. For the experiments with more than two decision classes, an averaged precision/recall is given in the cell, according to the description in Section 4.1.4. On the diagonal, when the test and training system are the same, the training data was derived from the first 70% revisions of the system evolution and the test data was built from the remaining 30%. The caption of each table explains what types of design anti-patterns were equated. The classifiers were trained to detect occurrences of various types of design anti-patterns or sets of these, if they were equated according to the description in Section 6.6.3. To

Pattern name	Metrics Filter	Argo UML	Elasticsearch	Struts	JHotDraw	Lucene	Wildfly	Xerces
Data Clump	$NOA > 3$	5%	7%	5%	5%	< 1%	6%	7%
Swiss Army Knife	$NOM > 6$, $MLOC > 150$	2%	1%	2%	2%	< 1%	< 1%	5%
Blob	$CYCL > 55$, $FA > 7$	2%	2%	2%	2%	2%	< 1%	6%
YoYo	$DIT > 5$	< 1%	2%	1%	< 1%	< 1%	< 1%	2%
Brain Class	$CLOC > 150$, $NOM > 15$, $FA > 5$	10%	7%	12%	10%	4%	3%	18%
Anemic Entity	$NOM > 8, NOF > 8$,	12%	9%	3%	13%	6%	9%	18%

Table A.12: Selectivity of metrics-based filter to find upper approximation of popular anti-patterns as an example of graph-index search. NOA = *Number Of Arguments*, NOM = *Number Of Methods*, NOF = *Number Of Fields*, MLOC = *Lines Of Code in Method(s)*, CYCL = *Cyclomatic complexity*, FA = *Fan out*, CLOC = *Lines Of Code in Class.*

reproduce these results run class EvolutionSplitSpatioTemporalRulesPredictionQualityExperiment with parameters set according to the following rules:

- To check prediction quality of a single specific type of design anti-pattern (Tables A.13 - A.20), set parameter DECISION_MAPPING_MODE to the name of the pattern. For example, to check the prediction quality of Blob, set DECISION_MAPPING_MODE="Blob".

- To check prediction quality when all types of design anti-patterns are equated(Table A.21), set parameter DECISION_MAPPING_MODE= "any-pattern",

- To check prediction quality when no types of design anti-patterns are equated(Table A.22), set parameter DECISION_MAPPING_MODE= "differentiate",

- To check prediction quality when some types of design anti-patterns are equated, set parameter DECISION_MAPPING_MODE="combine", and set parameter PARAM_DECISION_CLASS_COMBINATION to a string where types of design anti-patterns to equate are separated by # and separate decision classes are separated by ;. For example, if you want to equate Blob with YoYo and, independently, Base Bean with Data Clumps, set

train. data \ test data	argouml	xerces	struts	elastics- earch	jhotdraw	lucene- solr	wildfly
argouml	N/A	N/A	N/A	N/A	N/A	N/A	N/A
xerces	N/A	1.0/ 0.4	1.0/0.01	1.0/0.67	0/ 0.0	N/A	0.0/ 0.0
struts	N/A	0/ 0.0	0/ 0.0	0/ 0.0	0/ 0.0	N/A	0/ 0.0
elasticsearch	N/A	1.0/0.99	1.0/0.97	1.0/0.99	1.0/0.98	N/A	1.0/ 1.0
jhotdraw	N/A	0/ 0.0	0/ 0.0	0/ 0.0	0/ 0.0	N/A	0/ 0.0
lucene-solr	N/A	N/A	N/A	N/A	N/A	N/A	N/A
wildfly	N/A	0/ 0.0	0/ 0.0	0/ 0.0	0/ 0.0	N/A	0/ 0.0

Table A.13: Quality of prediction of design anti-pattern ParamDataClump.

train. data \ test data	argouml	xerces	struts	elastic- search	jhotdraw	lucene- solr	wildfly
argouml	1.0/0.78	1.0/ 0.0	0.8/0.04	1.0/0.94	1.0/0.28	1.0/0.93	1.0/0.91
xerces	0.99/0.95	1.0/0.89	1.0/0.71	1.0/0.94	1.0/0.96	1.0/0.93	1.0/0.91
struts	0.99/0.94	0.8/ 0.0	1.0/0.93	0.99/0.94	1.0/0.02	1.0/0.93	1.0/0.91
elasticsearch	1.0/ 0.5	1.0/0.42	1.0/0.28	1.0/ 0.7	1.0/ 0.9	1.0/0.59	1.0/0.92
jhotdraw	0/ 0.0	0/ 0.0	0/ 0.0	0/ 0.0	0/ 0.0	0/ 0.0	0/ 0.0
lucene-solr	1.0/0.96	1.0/0.98	1.0/0.93	0.99/0.95	1.0/0.96	0.99/ 0.9	1.0/0.91
wildfly	1.0/0.46	1.0/0.22	1.0/0.23	0.99/0.16	0.5/0.01	1.0/0.42	1.0/0.77

Table A.14: Quality of prediction of design anti-pattern SwissArmyKnifePattern.

PARAM_DECISION_CLASS_COMBINATION=
"Blob#Yoyo;BaseBean#DataClump".

Please note that according to the scheme described in B.6, you first need to run class MineSpatioTemporalRulesExperiment with the same parameters, to be able to run class EvolutionSplitSpatioTemporalRulesPredictionQualityExperiment (i.e. output of the former class must be stored in a file so that the latter class can read this file).

train. data \ test data	argouml	xerces	struts	elastic- search	jhotdraw	lucene- solr	wildfly
argouml	1.0/0.99	N/A	N/A	0.99/0.98	N/A	N/A	1.0/0.93
xerces	N/A	N/A	N/A	N/A	N/A	N/A	N/A
struts	N/A	N/A	N/A	N/A	N/A	N/A	N/A
elasticsearch	1.0/0.97	N/A	N/A	1.0/0.99	N/A	N/A	1.0/0.93
jhotdraw	N/A	N/A	N/A	N/A	N/A	N/A	N/A
lucene-solr	N/A	N/A	N/A	N/A	N/A	N/A	N/A
wildfly	0.99/0.92	N/A	N/A	1.0/0.95	N/A	N/A	1.0/0.84

Table A.15: Quality of prediction of design anti-pattern AnemicEntityPattern.

train. data \ test data	argouml	xerces	struts	elastic-search	jhotdraw	lucene-solr	wildfly
argouml	1.0/0.78	N/A	N/A	1.0/0.94	N/A	1.0/0.93	1.0/0.91
xerces	N/A	N/A	N/A	N/A	N/A	N/A	N/A
struts	N/A	N/A	N/A	N/A	N/A	N/A	N/A
elasticsearch	1.0/ 0.5	N/A	N/A	1.0/ 0.7	N/A	1.0/0.59	1.0/0.92
jhotdraw	N/A	N/A	N/A	N/A	N/A	N/A	N/A
lucene-solr	1.0/0.96	N/A	N/A	0.99/0.95	N/A	0.99/ 0.9	1.0/0.91
wildfly	1.0/0.46	N/A	N/A	0.99/0.16	N/A	1.0/0.42	1.0/0.77

Table A.16: Quality of prediction of design anti-pattern BaseBeanPattern.

train. data \ test data	argouml	xerces	struts	elastic-search	jhotdraw	lucene-solr	wildfly
argouml	1.0/0.76	1.0/0.74	N/A	1.0/0.59	1.0/0.16	0.99/0.45	1.0/0.45
xerces	1.0/ 0.5	1.0/0.77	N/A	1.0/0.59	1.0/0.16	0.99/0.45	1.0/0.41
struts	N/A	N/A	N/A	N/A	N/A	N/A	N/A
elasticsearch	0.99/0.48	1.0/0.74	N/A	0.99/0.75	0.92/0.15	1.0/0.55	1.0/0.32
jhotdraw	0.98/0.58	0.99/0.74	N/A	0.99/0.59	1.0/0.15	0.99/0.45	1.0/ 0.6
lucene-solr	1.0/0.26	1.0/0.76	N/A	1.0/0.79	1.0/0.18	1.0/0.69	1.0/0.26
wildfly	0.96/0.33	1.0/0.24	N/A	0.99/0.59	1.0/0.87	1.0/0.54	1.0/ 0.8

Table A.17: Quality of prediction of design anti-pattern BlobPattern.

train. data \ test data	argouml	xerces	struts	elastic-search	jhotdraw	lucene-solr	wildfly
argouml	1.0/ 0.7	1.0/0.65	N/A	1.0/0.52	N/A	1.0/0.86	N/A
xerces	1.0/0.87	0.99/0.65	N/A	1.0/0.79	N/A	1.0/0.18	N/A
struts	N/A	N/A	N/A	N/A	N/A	N/A	N/A
elasticsearch	0.99/0.62	0.99/0.34	N/A	1.0/0.74	N/A	0.99/0.78	N/A
jhotdraw	N/A	N/A	N/A	N/A	N/A	N/A	N/A
lucene-solr	1.0/0.47	0.99/0.74	N/A	1.0/0.83	N/A	0.99/0.73	N/A
wildfly	N/A	N/A	N/A	N/A	N/A	N/A	N/A

Table A.18: Quality of prediction of design anti-pattern YoYoPattern.

train. data \ test data	argouml	xerces	struts	elastic-search	jhotdraw	lucene-solr	wildfly
argouml	N/A	N/A	N/A	N/A	N/A	N/A	N/A
xerces	N/A	0.99/0.98	N/A	1.0/0.01	N/A	0.8/ 0.0	1.0/0.96
struts	N/A	N/A	N/A	N/A	N/A	N/A	N/A
elasticsearch	N/A	1.0/0.98	N/A	1.0/0.99	N/A	1.0/0.98	1.0/0.96
jhotdraw	N/A	N/A	N/A	N/A	N/A	N/A	N/A
lucene-solr	N/A	1.0/ 0.4	N/A	1.0/0.97	N/A	1.0/0.87	1.0/0.96
wildfly	N/A	1.0/0.01	N/A	0.95/0.03	N/A	1.0/0.03	1.0/0.19

Table A.19: Quality of prediction of design anti-pattern CircularDependencyPattern.

train. data \ test data	argouml	xerces	struts	elastic-search	jhotdraw	lucene-solr	wildfly
argouml	N/A	N/A	N/A	N/A	N/A	N/A	N/A
xerces	N/A	N/A	N/A	N/A	N/A	N/A	N/A
struts	N/A	N/A	N/A	N/A	N/A	N/A	N/A
elasticsearch	N/A	N/A	N/A	1.0/0.96	1.0/0.92	1.0/0.92	N/A
jhotdraw	N/A	N/A	N/A	0.99/0.96	1.0/0.93	1.0/0.92	N/A
lucene-solr	N/A	N/A	N/A	1.0/0.17	1.0/0.93	1.0/0.55	N/A
wildfly	N/A	N/A	N/A	N/A	N/A	N/A	N/A

Table A.20: Quality of prediction of design anti-pattern BrainClassPattern.

train. data \ test data	argouml	xerces	struts	elastic-search	jhotdraw	lucene-solr	wildfly
argouml	0.97/0.75	0.99/0.72	1.0/ 1.0	0.99/0.98	1.0/ 1.0	1.0/ 1.0	0.99/0.85
xerces	0.99/0.67	0.99/0.93	0.94/0.65	0.99/0.84	0.9/0.11	0.99/0.63	0.97/0.17
struts	1.0/ 1.0	0.99/ 1.0	0.96/0.74	0.98/0.25	0.0/ 0.0	0.98/0.13	1.0/ 1.0
elasticsearch	0.99/0.67	0.99/0.79	0.98/0.76	0.99/0.88	0.9/0.11	0.99/0.56	0.97/0.17
jhotdraw	0.96/0.35	0.97/0.24	0.57/0.04	0.99/0.99	0.5/0.04	0.99/ 1.0	0.99/0.96
lucene-solr	0.99/0.98	0.98/0.42	0.98/0.98	0.99/0.97	1.0/ 1.0	0.99/0.97	0.99/ 0.9
wildfly	1.0/ 1.0	0.99/ 1.0	1.0/ 1.0	1.0/ 1.0	1.0/ 1.0	0.99/ 1.0	1.0/ 1.0

Table A.21: Quality of prediction of instances of any design anti-pattern (types of design anti-patterns are indiscernible).

train. data \ test data	argouml	xerces	struts	elastic-search	jhotdraw	lucene-solr	wildfly
argouml	0.97/0.76	0.99/0.75	1.0/0.53	0.95/0.03	1.0/0.16	0.98/0.19	1.0/0.19
xerces	0.95/0.09	1.0/0.98	1.0/0.38	1.0/0.97	0.98/0.94	1.0/0.07	0.91/0.05
struts	1.0/0.15	1.0/0.98	1.0/0.91	0.99/0.73	1.0/0.38	0.99/0.95	1.0/0.11
elasticsearch	0.98/0.36	1.0/0.35	0.97/0.45	0.99/0.75	0.97/0.83	0.99/0.57	1.0/0.64
jhotdraw	0.5/ 0.0	0.99/ 0.6	1.0/0.76	0.9/0.02	1.0/0.18	0.92/0.03	1.0/0.54
lucene-solr	0.95/0.17	1.0/0.32	0.96/0.38	0.99/0.42	0.5/0.01	0.99/0.44	0.99/0.89
wildfly	0.8/0.07	0.57/ 0.0	0.0/ 0.0	0.9/0.02	0.75/0.03	0.87/0.03	0.85/0.09

Table A.22: Quality of prediction of different design anti-patterns (each type of design anti-pattern is a separate decision class).

Dataset	Pattern	Spatial Precision/Recall	Spatio-temporal Precision/Recall	Change of F1
elastic	BaseBean	0.71 / 0.88	1.0 / 0.54	0.89
argouml	BLOB	0.9 / 0.76	1.0 / 0.76	1.05
elastic	BLOB	0.83 / 0.9	1.0 / 0.88	1.08
Xerces	BLOB	0.91 / 0.69	1.0 / 0.69	1.04
elastic	BrainClass	0.87 / 0.84	1.0 / 0.81	1.05
argouml	SAK	0.78 / 1.0	1.0 / 0.94	1.11
elastic	SAK	0.78 / 0.99	1.0 / 0.99	1.14
Lucene	SAK	0.86 / 1.0	1.0 / 0.92	1.04
Xerces	SAK	0.8 / 0.89	1.0 / 0.65	0.94
Xerces	YoYo	0.98 / 1.0	1.0 / 0.99	1.01
			Average	1.04

Table A.23: Impact of spatio-temporal rules on static prediction quality

A.6.2 Static pattern detection improved with spatio-temporal rules

Section 6.7.4 explains how spatio-temporal rules can improve detection quality of static patterns. Table A.23 shows the cases, when the result of detection was different. In two cases the quality decreased by 6-11%, and in all other cases it improved by 1-14% in terms of F1. There was an average improvement of 4%. To reproduce the experiment run class FindStaticPatternsExperiment with input property $INCLUDE\_SPATIO\_TEMPORAL\_RULES = true$

A.7 Attractors and repellents

Table A.24 shows the attraction coefficient between different types of design anti-patterns, defined in Section 6.7.4. To reproduce this result, run class AttractorsAndRepellentsExperiment.

Pattern name	DataCl	SAK	AnEnt	BasBe	Blob	YoYo	CircDep	BrainCl
DataCl		N/A	+0,20	N/A	N/A	-0,14	+0,05	+0,10
SAK			+0,20	N/A	N/A	-0,14	+0,05	+0,10
AnEnt				+0,20	+0,20	+0,63	+0,60	+0,67
BasBe					N/A	-0,14	+0,05	+0,10
Blob						-0,14	+0,05	+0,10
YoYo							+0,78	-0,32
CircDep								+0,77
BrainCl								

Table A.24: *Attraction coefficient* (see Section 6.7.4) between different types of design anti-patterns. DataCl = DataClump, SAK = SwissArmyKnife, AnEnt = AnemicEntity, BasBe = BaseBean, CircDep = CircularDependency, BrainCl = BrainClass,

Appendix B

Experimental reproduction

B.1 Expert tagging

In order to be able to determine the quality of mining design anti-pattern in given software, we need to have reference data with all its actual instances in the analyzed software. This data is frequently provided by an external oracle, which typically is a human expert. Such expert tagging used in this research is available in [279] and was derived from openly available data of this kind (see [323]): [185], [269], [267], [221], [242], [184].

The expert data was constructed from the mentioned data in such a way that a subgraph was considered an instance of a respective design anti-pattern if it was marked so in at least one of the available data sets. Only when a certain type was not covered in any of the original data sets (e.g. YoYo), then such instances were manually appointed by an expert. It must be mentioned that in [241], which is research bound to data [242], the authors imply that if a certain structure of software appears to be a code smell, but it is required by the external library, e.g. implementation of Java SAX parser (see [261]), it should not be considered an instance of respective code smell. Contrary to that, in our research such cases were not excluded from expert tagging, since we believe that a structure should be considered an instance of a design anti-pattern according to its immanent nature (see [122]). However, we excluded the source files that are automatically generated by a script (e.g. language stemmers in lucene-solr), as this kind of files are not created by developers.

B.2 Source code

The source code of the experiments conducted in the course of this thesis is available at [281]. File readme.md from this archive provides information on how to run it.

B.3 Environment for experiments

This section describes technical environment required to reproduce the results presented in this thesis. All experiments described were conducted on Windows 10 with Java JDK 11. All git SCM repositories must be cloned to a common folder pointed by system variable REPO_DIR (e.g. if %REPO_DIR%=c:\repos, then Xerces scm must be cloned to c:\repos\xerces, argouml must be cloned to c:\repos\argouml, etc.)
A script setEnv.bat available at [280] clones all data used in this research to %REPO_DIR% folder (see Section 6.7.1). Expert tagging for static patterns detection described in Sections 6.4 and 6.7.4 must be located in a folder pointed by system variable DATA_DIR. Again, each analyzed system will have in it one dedicated sub-folder. The reference contents of all sub-folders are given in [279]. Two additional folders are required to store preprocessed data necessary for indirect use in many experiments. A system variable EVOLUTIONS_DIR must point to the folder where software evolutions, described in Section 6.1, will be stored in the form of XML files. Detailed description of this mechanism is given in Section B.5. Each sub-folder of %EVOLUTIONS_DIR% must contain information about the evolution of a single system. The logs of SCM of each system are translated into XML representation of SSn iteratively, revision by revision. Since this process may be time-consuming, it may be interrupted at each revision and then resumed from the same revision. Therefore, each sub-folder of EVOLUTIONS_DIR also contains a configuration file with information necessary to resume the process. The template of such configuration is available at [277]. A system variable TEMP_WORKING_DIR must point to a temporary folder where data passed between dependent experiments will be stored in dedicated files. No sub-folders or files are required in this folder. Because of large amounts of data, it is required that volumes in which EVOLUTIONS_DIR and TEMP_WORKING_DIR are located must have at least 100GB of free space.

B.4 Running the experiments

This section provides general remarks on running the experiments implemented in the source code mentioned above in Section B.2. Specific remarks required to run an individual experiment are included in the section devoted to this experiment in Appendix A.

Each experiment described in this thesis has a corresponding java class that implements interface Experiment, which defines one method:

```
void run(ExperimentInput input, ExperimentOutput output)
```

In order to reproduce the experiment one has to invoke this method on an instance of the respective class. The first argument of this method is used to pass input parameters, which control the experiment. A description of parameters specific to a given experiment is given in the javadoc of the respective class (see [278]). Exemplary code of running an experiment with one parameter is given in Listing B.1.

Listing B.1: Example: running an experiment with one parameter

```java
//experiment configuration - setting the parameters
ExperimentInput input = new ExperimentInput();
input.setProperty(PARAM_INCLUDE_SPATIO_TEMPORAL_RULES, true);
input.setProperty(EVOLUTION_NAME_PROP, "lucene-solr");

//running the experiment
ExperimentOutput output = new ExperimentOutput();
new FindStaticPatternsExperiment().run(input, output);
```

B.5 Software evolution serialization

The initial experiment to run is SerializeHistoryExperiment. It fetches logs from SCM and issue tracker of a given system and it translates into XML representation of ssn. We will call this process *software evolution serialization.* The produced XML file will be used as the source of data for most of the other experiments. Since this computation is time- and resource-consuming, it should be executed for each analyzed system separately. Please note that serialization is equivalent to running the FP function defined in Section 6.2.2.

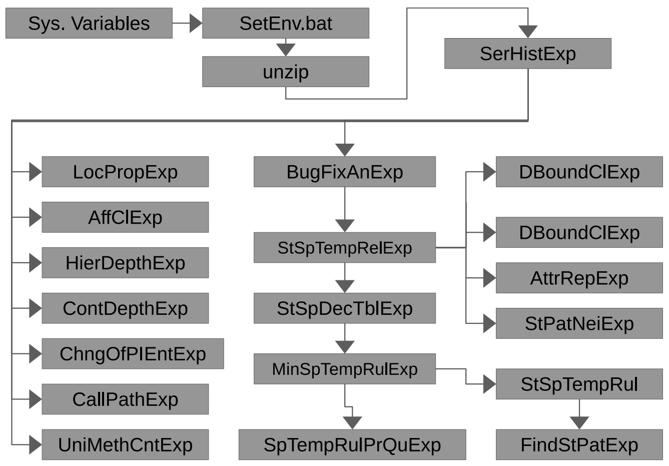

Figure B.1: Dependencies between experiments. Reproduction of experiments should be executed in the order defined by the arrows. Sys. Vari, setEnv.bat, unzip – see Section B.3. SerHistExp = class SerializeHistoryExperiment, StSpTempRelExp = class StoreSpatioTemopralRelationsExperiment, MinSpTempRulExp = class MineSpatioTemporalRulesExperiment, AttrRepExp = class AttractorsAndRepellentsExperiments, LocPropExp = class LocalityPropertiesExperiment, UniMethCntExp = class UninomialMethodsCountExperient, AffClExp = class AffectedClassesSizeExperiment, StPatNeiExp = class StaticPatternsNeighborhoodSizeExperiment, ChngOfPIEntExp = class ChangeOfPatternInstanceEntitiesSizeExperiment, DBoundClExp = class SizeOfGraphBoundedByPathStartingAtAffectedEntitiesExperiment, SpTempRulPrQuExp = class EvolutionSplitSpatioTemporalRulesPredictionQualityExperiment, HierDepthExp= class HiearchyDepthDitributionExperiment, ContDepthExp = class ContainmentDepthDistributionExperiment, CallPathExp = class CallDepthDistributionExperiment, BugFixAnExp = class BugfixAnalysisExperiment, StSpDecTblExp = class StoreSpatioTemporalDecisionTableExperiment, AttrRepExp = class AttractorsAndRepellentsExperiment, FindStPatExp = class FindStaticPatternsExperiment, StSpTempRul class StoreSpatioTemporalRulesForStaticPatternDetectionExperiment

B.6 Dependencies between experiments

Figure fig:Dependencies of experiment outlines the dependencies between different experiments. By convention arrows show precedence in the order of experiments, i.e. the item pointed by an arrow can only be run after all other items connected to the other end of the arrow were run previously.