Development of Agent-Based Simulation Models for Software Evolution

Dissertation
zur Erlangung des mathematisch-naturwissenschaftlichen Doktorgrades
"Doctor rerum naturalium"
der Georg-August-Universität Göttingen

im Promotionsprogramm Computer Science (PCS)
der Georg-August University School of Science (GAUSS)

vorgelegt von

Daniel Honsel
aus Hildesheim

Göttingen, 2019

Tag der mündlichen Prüfung
12.12.2019

Abstract

Software has become a part of everyday life for us. This is also associated with increasing requirements for adaptability to rapidly changing environments. This evolutionary process of software is being studied by a software engineering related research area, called software evolution. The changes to a software over time are caused by the work of the developers. For this reason, the developer contribution behavior is central for analyzing the evolution of a software project. For the analysis of real projects, a variety of open source projects is freely available. For the simulation of software projects, we use multiagent systems because this allows us to describe the behavior of the developers in detail.

In this thesis, we develop several successive agent-based models that cover different aspects of software evolution. We start with a simple model with no dependencies between the agents that can simulative reproduce the growth of a real project solely based on the developer's contribution behavior. Subsequent models were supplemented by additional agents, such as different developer types and bugs, as well as dependencies between the agents. These advanced models can then be used to answer different questions concerning software evolution simulative. For example, one of these questions answers what happens to the software in terms of quality when the core developer suddenly leaves the project. The most complex model can simulate software refactorings based on graph transformations. The simulation output is a graph which represents the software. The representative of the software is the change coupling graph, which is extended for the simulation of refactorings. In this thesis, this graph is denoted as *software graph*.

To parameterize these models, we have developed different mining tools. These tools allow us to instantiate a model with project-specific parameters, to instantiate a model with a snapshot of the analyzed project, or to parameterize the transformation rules required to model refactorings.

The results of three case studies show, among other things, that our approach to use agent-based simulation is an appropriate choice for predicting the evolution of software projects. Furthermore, we were able to show that different growth trends of the real software can be reproduced simulative with a suitable selection of simulation parameters. The best results for the simulated software graph are obtained when we start the simulation after an initial phase with a snapshot of real software. Regarding refactorings, we were able to show that the model based on graph transformations is applicable and that it can slightly improve the simulated growth.

Zusammenfassung

Software ist ein Bestandteil des alltäglichen Lebens für uns geworden. Dies ist auch mit zunehmenden Anforderungen an die Anpassungsfähigkeit an sich schnell ändernde Umgebungen verbunden. Dieser evolutionäre Prozess der Software wird von einem dem Software Engineering zugehörigen Forschungsbereich, der Softwareevolution, untersucht. Die Änderungen an einer Software über die Zeit werden durch die Arbeit der Entwickler verursacht. Aus diesem Grund stellt das Entwicklerverhalten einen zentralen Bestandteil dar, wenn man die Evolution eines Softwareprojekts analysieren möchte. Für die Analyse realer Projekte steht eine Vielzahl von Open Source Projekten frei zur Verfügung. Für die Simulation von Softwareprojekten benutzen wir Multiagentensysteme, da wir damit das Verhalten der Entwickler detailliert beschrieben können.

In dieser Dissertation entwickeln wir mehrere, aufeinander aufbauende, agentenbasierte Modelle, die unterschiedliche Aspekte der Software Evolution abdecken. Wir beginnen mit einem einfachen Modell ohne Abhängigkeiten zwischen den Agenten, mit dem man allein durch das Entwicklerverhalten das Wachstum eines realen Projekts simulativ reproduzieren kann. Darauffolgende Modelle wurden um weitere Agenten, zum Beispiel unterschiedliche Entwickler-Typen und Fehler, sowie Abhängigkeiten zwischen den Agenten ergänzt. Mit diesen erweiterten Modellen lassen sich unterschiedliche Fragestellungen betreffend Software Evolution simulativ beantworten. Eine dieser Fragen beantwortet zum Beispiel was mit der Software bezüglich ihrer Qualität passiert, wenn der Hauptentwickler das Projekt plötzlich verlässt. Das komplexeste Modell ist in der Lage Software Refactorings zu simulieren und nutzt dazu Graph Transformationen. Die Simulation erzeugt als Ausgabe einen Graphen, der die Software repräsentiert. Als Repräsentant der Software dient der Change-Coupling-Graph, der für die Simulation von Refactorings erweitert wird. Dieser Graph wird in dieser Arbeit als *Softwaregraph* bezeichnet.

Um die verschiedenen Modelle zu parametrisieren haben wir unterschiedliche Mining-Werkzeuge entwickelt. Diese Werkzeuge ermöglichen es uns ein Modell mit projektspezifischen Parametern zu instanziieren, ein Modell mit einem Snapshot des analysierten Projektes zu instanziieren oder Transformationsregeln zu parametrisieren, die für die Modellierung von Refactorings benötigt werden.

Die Ergebnisse aus drei Fallstudien zeigen unter anderem, dass unser Ansatz agentenbasierte Simulation für die Vorhersage der Evolution von Software Projekten eine geeignete Wahl ist. Des Weiteren konnten wir zeigen, dass mit einer geeigneten Parameterwahl unterschiedliche Wachstumstrends der realen Software simulativ reproduzierbar sind. Die besten Ergebnisse für den simulierten Softwaregraphen erhalten wir, wenn wir die Simulation nach einer initialen Phase mit einem Snapshot der realen Software starten. Die Refactorings betreffend konnten wir zeigen, dass das Modell basierend auf Graph Transformationen anwendbar ist und dass das simulierte Wachstum sich damit leicht verbessern lässt.

Acknowledgements

I would like to thank several persons who supported me during my work on this thesis. First, I want to thank my first supervisor Prof. Dr. Stephan Waack who gave me the opportunity to focus my research on the exiting topic of agent-based modeling and simulation. He was always available for fruitful and interesting discussions.

Also, I would like to thank my second supervisor Prof. Dr. Jens Grabowski for providing valuable feedback and discussions concerning my work, especially the software engineering part of this thesis. Moreover, I want to thank the thesis committee members Prof. Dr.-Ing. Marcus Baum, Prof. Dr. Carsten Damm, Prof. Dr. Florin Manea, and Prof. Dr. Kerstin Strecker for spending their precious time.

Furthermore, many thanks to my current and former colleagues in my research group and at the institute for interesting discussions and for providing an enjoyable environment to work. Especially, I want to thank Linh Dangh for proofreading this thesis. Moreover, I would like to thank Dr. Steffen Herbold and Dr. Fabian Trautsch for supporting me in developing some of the mining tools for this thesis.

In addition, I thank the SWZ Clausthal-Göttingen[1] that partially funded our work in the projects "Simulation-based Quality Assurance for Software Systems" and "Agent-based simulation models in support of monitoring the quality of software projects". Many thanks also to all former members of these projects for valuable discussions and a pleasant cooperation.

Especially I would like to thank my colleague and sister Verena for a successful cooperation in our projects and for all her support, and for proofreading this thesis.

I also want to thank my parents. They have always supported my decisions concerning my education and career.

Very special thanks to my girlfriend Anika Werner who has not stopped motivating and supporting me during my work on this thesis. Moreover, I would like to thank her for proofreading this thesis.

Finally, I would like to thank our cats Gimli and Balu for some necessary breaks and distractions at exactly the right time.

[1]https://www.simzentrum.de/en/

Contents

1. Introduction

Contents

At the present time, many people are in contact with software in their everyday lives. This begins, for example, with the smartphone, which is used for more and more everyday tasks, goes through traffic planning and train timetables to software installed at the computer at home.

All these software systems evolve over time due to changing requirements, changing environments, or some required maintenance work. This is where the prominent research area of software evolution comes in. Software Evolution is integrated into the field of software engineering and deals with the analysis of the process of software projects. For this, the past of a software project can be considered to predict the future progress of the analyzed project. As far as the actual software is concerned, the state of the software mainly depends on the contribution behavior of the developers involved in the project. This behavior is responsible for software changes over the time. Developers can be divided into different types. These types differ, for example, in their contribution behavior and their commit frequency [1]. In order to build a predictive model for software evolution, all these facets must be gathered from the project to analyze.

To analyze the past of a software system, real project data is required. Because there are more and more open source projects hosted on platforms like github, there is a lot of data for a variety of projects free available. The data retrieved by mining some of these projects is used to estimate parameters for a simulation model that predicts the future progress of the projects. For this, the commit history as well as the source code are analyzed. Furthermore, change coupling graphs are considered to represent sematic relationships between files [2]. We developed several mining tools which, for example, instantiate a given simulation model with a project specific set of parameters or provide parameters to instantiate a simulation model at any desired point in time of the past of the analyzed project. Furthermore, a mining tool to find and parameterize commit pattern for applied refactorings is developed.

For the simulation of software evolution, that predicts the future of the analyzed project, multiagent systems [3] are used. In such a system the behavior of the agents make the entire system evolve over time. Since software evolution is mainly influenced by the developer's behavior, we think Agent-Based Modeling and Simulation (ABMS) is well suited for this simulation purpose. Besides that, a detailed description of the individual agents and their behavior is required. We developed several simulation models that evolve step by step by adding more agent types or dependencies between the agents in each step. Each of these steps has a specific goal, such as the generation of the change coupling graph with the simulation or answering further research questions regarding software evolution.

This simulation model can be used by a project manager to answer various questions regarding the quality of the analyzed software. These questions may, for example, concern changes in the constellation of developers involved in a project or the lifetime of bugs. To answer the question of a manager, a feedback loop can be used. This means that several simulation runs are performed with different parameters until the result meets the expectations of the manager. In order to be able to answer these questions realistically, the simulated software graph must behave similarly to the realistic software graph. To validate this, selected graph metrics of the simulated graph are compared with the corresponding real graph metrics.

1.1. Scope of the Thesis

We want to figure out whether it is possible to simulate evolving software systems using ABMS. The goal is to answer research questions concerning software evolution as well as to generate realistic change coupling graphs as simulation output. Therefore, we developed models that should answer specific questions and compared the simulated with the real change coupling graph of selected open source projects.

The first case study investigates which aspects of software evolution can be simulated using a certain kind of simulation model. The models differ in the number of different agent types that are involved as well as in the modeled dependencies to describe relationships between the agents. Parameters for different projects come partly from a reference project and partly from the simulated project. We found that a model without any modeled dependencies can simulate the growth of a project [4]. Furthermore, more complex models can be used to answer questions like: Can we simulate the effects when a core developer leaves the project [5]?

The topic of the second case study is the quality of the simulated change coupling graph. In order to make a statement about this, we compared selected graph metrics of the simulated graph with the real graph. Compared metrics are, for example, the number of nodes, the average degree of the nodes, the density of the graph, or the diameter of the graph. For the comparison, we have designed two different scenarios. First, the simulation model is instantiated with project specific parameters for each project to analyze and the simulation starts at the beginning of the project. Second, the simulation model is initialized with project specific parameters as well as the change coupling graph of a given year. Afterwards, the simulation starts at this point in time. Our main findings are that we can reproduce different growth types of the software with the project specific parameters and that metrics of the simulated graph fits the real metrics when the simulation is initialized with parameters starting approximately after one third of the project duration.

The third case study is about the mining and simulation of software refactorings [6]. With refactorings we can model the intention of developers and consider more aspects concerning the quality of the evolving project. We want to show that we can retrieve parameters for a simulation model that uses graph transformation rules for the description of software refactorings [7]. Furthermore, we consider the impact of such an extended simulation model on the quality of the simulated change coupling graph. We figured out that the simulation of refactorings using graph transformations works and that the growth trend of a project can be slightly improved when refactorings are simulated.

1.2. Thesis Impact

This work is part of the two SWZ projects *Simulation-Based Quality Assurance for Software Systems*[1] and *Agent-based simulation models in support of monitoring the quality of software projects*[2].

During this work, the following papers have been published in peer reviewed conference proceedings:

- Daniel Honsel, Niklas Fiekas, Verena Herbold, Marlon Welter, Tobias Ahlbrecht, Stephan Waack, Jürgen Dix, Jens Grabowski, "Simulating Software Refactorings based on Graph Transformations", in *Post-Proceedings of the Clausthal-Göttingen International Workshop on Simulation Science 2017*, Springer, 2018

[1]https://www.simzentrum.de/en/education/softwarequalitaetssicherung-mit-hilfe-von-simulationsverfahren

[2]https://www.simzentrum.de/en/research-projects/agent-based-simulation-models-in-support-of-monitoring-the-quality-of-software-projects

Own contributions
I am the lead author of the paper. I contributed significantly to the design of the approach, the mining process and the evaluation of the approach. The used simulation framework is developed by N. Fiekas.

- Daniel Honsel, Verena Honsel, Marlon Welter, Jens Grabowski, Stephan Waack, "Monitoring Software Quality by Means of Simulation Methods", in *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM 2016)*, short paper, 2016

  Own contributions
  I am the lead author of the paper. I contributed significantly to the simulation model including different behavior strategies for the agents and its implementation. Required simulation parameters are mined by V. Honsel. Furthermore, the conceptual work and the case study design was joined work with V. Honsel. The automated assessment of software graphs was done by M. Welter.

Furthermore, some papers were published to which the author of this thesis contributed:

- Marlon Welter, Daniel Honsel, Verena Herbold, Andre Staedler, Jens Grabowski, Stephan Waack, "Assessing Simulated Software Graphs using Conditional Random Fields", in *Post-Proceedings of the Clausthal-Göttingen International Workshop on Simulation Science 2017*, Springer, 2018

  Own contributions
  Own contributions for this paper include some conceptual work for the generation of required software graphs. The CRF assessment tool is developed and evaluated by M. Welter.

- Tobias Ahlbrecht, Jürgen Dix, Niklas Fiekas, Jens Grabowski, Verena Herbold, Daniel Honsel, Stephan Waack, Marlon Welter, "Agent-based simulation for software development processes", on *Proceedings of the 14th European Conference on Multi-Agent Systems (EUMAS 2016)*, Springer, 2016

  Own contributions
  Own contributions to this paper are the modeling and implementation of the non distributed version of the simulation model for software evolution. Furthermore, I was involved in the design of the proposed approach. The parameter ming for the simulation model is done by V. Herbold. The distributed simulation framework is provided by T. Ahlbrecht and N. Fiekas.

- Verena Honsel, Daniel Honsel, Steffen Herbold, Jens Grabowski, Stephan Waack, "Mining Software Dependency Networks for Agent-Based Simulation of Software Evolution", in *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW), The 4th International Workshop on Software Mining*, 2015

Own contributions
Own contributions to this paper include the design and the implementation of the required simulation model. Furthermore, I was involved in the design of the proposed approach concerning the determination of required simulation parameters. The mining process and the evaluation of the approach is provided by V. Honsel.

- Verena Honsel, Daniel Honsel, Jens Grabowski, Stephan Waack, "Developer Oriented and Quality Assurance Based Simulation of Software Processes", in *Proceedings of the Seminar Series on Advanced Techniques & Tools for Software Evolution (SATToSE 2015)*, 2015

  Own contributions
  This paper presents a summary of the papers [4], [8], and [9]. Thus, it is joined work of all involved authors. Own contributions include parts the conceptual work and parts of the summary of the considered papers.

- Verena Honsel, Daniel Honsel, Jens Grabowski, "Software Process Simulation based on Mining Software Repositories", in *Proceedings of the IEEE International Conference on Data Mining Workshop (ICDM 2014)*, short paper, 2014

  Own contributions
  The own contribution to this paper is the modeling and implementation of the agent-based simulation model. Furthermore, the evaluation of the simulated data was joined work with V.Honsel. The mining process and analysis of mined data is provided by V.Honsel.

In addition, the following book chapter has been published containing parts of the work developed in this thesis:

- Philip Makedonski, Verena Herbold, Steffen Herbold, Daniel Honsel, Jens Grabowski, Stephan Waack, "Mining Big Data for Analyzing and Simulating Collaboration Factors Influencing Software Development Decisions", in *Social Network Analysis: Interdisciplinary Approaches and Case Studies*, CRC Press, 2016

  Own contributions
  The own contribution to this book chapter is the adaption of the agent-based simulation model developed by the author of this thesis. The model has been modified to support the collaborative networks analyzed in this chapter. The mining of software projects to build developer social networks as well as the analysis of these is provided by V. Herbold. This work establish an example application for the fine-grained developer behavior and collaboration model presented in this book chapter by Dr. P. Makedonski.

1.3. Thesis Structure

This thesis has a focus on the development, the parametrization, and the validation of agent-based simulation models for software evolution. Following the introduction, the theoretical background of this thesis is described. Afterwards, related work is presented. Then, the developed simulation models are presented and required mining tools, in order to retrieve parameters for the model instantiation, are introduced. The simulation models are evaluated in three case studies which are discussed and summarized at the end of this thesis. The detailed content of the chapters is presented in the following.

- **Chapter 2 (Background)** describes the theoretical background of this thesis. Since the proposed approach in this thesis covers the research ares multiagent system, software evolution, software refactorings, graph transformations, and mining software repositories, all of them are introduced.
- **Chapter 3 (Related Work)** presents the latest state of the art in the covered research areas of this thesis.
- **Chapter 4 (Evolution of Agent-Based Simulation Models)** describes the evolution of an Agent-Based Model (ABM) for software processes. Starting with a model without dependencies between the agents to reproduce the growth of a software project, we motivate to introduce step by step more dependencies or agents to be able to answer more complex research questions or to improve the quality of the simulated change coupling graph. Furthermore, implementation details are presented and required parameters are described. Besides that, it is illustrated how the simulation application is adaptable at runtime by using different parameters.
- **Chapter 5 (The Gathering of Parameters for Model Execution)** introduces the developed mining frameworks of this thesis. The automated parameter estimation tool is required to initialize a simulation model with a complete set of project specific parameters as well as for the retrieval of the change coupling graph. This graph is used to initialize the model at a certain point in time as well as for validation purposes. Furthermore, tools to parameterize the refactoring model are presented.
- **Chapter 6 (Case Studies)** presents the three case studies of this thesis, each containing the setup, the results and a briefly discussion. The first case study evaluates the steps of the model evolution. The second case study compares the simulated change coupling graph with the real one of selected projects. Furthermore, the changes to the simulated graph for initialized models after one third of the project duration are analyzed. The third case study considers the feasibility of our approach to simulate refactorings and analyses how simulated refactorings change the simulated change coupling graph.

- **Chapter 7 (Discussion)** considers the results of all three case studies as a whole and discusses strength and limitations of this approach. At the end, the contribution is pointed out.
- **Chapter 8 (Conclusion)** summarizes this thesis and presents briefly the main findings. Finally, some future work based on this thesis is discussed.

2. Background

Contents

In this chapter, we present the foundations required to model the evolution of software processes using ABMS. The decision to use ABMS is justified by the fact that software evolution is generally based on the work of the participating developers. Thus, it seems to be natural to model software processes from the starting point of human behavior.

This chapter is structured as follows. In Section 2.1, we introduce multiagent systems, Section 2.2 explains the meaning of the term software evolution and presents evolving variables and data structures analyzed to model software evolution. In Section 2.3, we describe the meaning of the term refactoring. To model refactorings we use graph transformations which are introduced in Section 2.4. Finally, we present relevant data sources and our data retrieval process to parameterize the proposed model in Section 2.5.

2.1. Multiagent Systems

Multiagent systems are systems that contain multiple intelligent agents that interact with each other. An agent could be either a computational entity such as a software program or a robot. Situated in some environment an agent acts autonomously and self-directed to achieve its goal. Agents perceive their local environment and can make decisions without the intervention of humans or other systems solely based on the state of the environment and their behavior.

There exists a wide range of potential instantiations of concrete multiagent systems. A system consisting of multiple agents, interaction possibilities, and an environment can differ in the relevant attributes as shown in Table 2.1.

The definition of a multiagent system as well as the contents of Table 2.1 are based on Weiss at al. [3]. In the following section, we describe what agents are in more detail and we will introduce tools for ABMS. Especially the ABMS framework Repast Simphony [10], which is used for modeling and simulation purposes in this thesis, is discussed in more detail.

2.1.1. What are Agents?

To explain what is meant by the term *agent* we start this section with a definition. The following definition of the term *agent* is based on [3, 11, 12].

> ”An agent is a computer system that is situated in some environment, and that is capable of autonomous action in this environment in order to achieve its delegated objectives.”

	attribute	**range**
agents	number	from two upwards
	uniformity	homogeneous ... heterogeneous
	goals	contradicting ... complementary
	flexibility	reactive ... deliberative
	autonomy	low ... high
interaction	frequency	low ... high
	persistence	short-term ... long-term
	language	elementary ... semantically rich
	variability	fixed ... changeable
	purpose	competitive ... cooperative
environment	predictability	foreseeable ... unforeseeable
	accessibility	unlimited ... limited
	dynamics	fixed ... variable
	diversity	poor ... rich
	availability of resources	restricted ... ample

Table 2.1.: Combination possibilities of multiagent systems (adapted from [3]).

Agents that satisfy the definition are, for example, a simple thermostat system or a robot playing soccer. A thermostat can regulate the room temperature according to the measured data of the environment with only two actions, turn *on* and *off* the heating. The second example is a more complex one. The robot is situated in a labor environment as shown in Figure 2.1. The soccer field with its green ground and white lines, a red ball, and colored goals (according to the RoboCup Standard Platform League Rules of 2011 [13]). Based on its sensor data the robot makes decisions to achieve its objectives. The main objective of the robot is to score a goal. Therefore, possible actions could be *reach_the_ball* or *kick*. More about this domain can be found as an example application in Section 2.1.3. Both examples fit the definition of an agent, but only the robot example is what is called an *intelligent agent* according to [3]. The term *intelligent agent* will be briefly described in the following.

Intelligent Agents

Weiss et al. [3] defined *intelligent agents* as agents with the following additional behavior characteristics:

- *proactiveness:* intelligent agents are goal-directed, which means that they are taking the initiative in order to reach their defined goals;

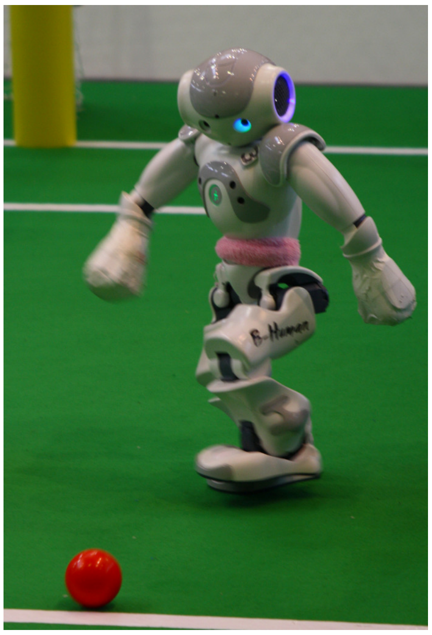


Figure 2.1.: NAO robot playing soccer [14].

- *reactivity:* intelligent agents are able to perceive their environment and can react on changes according to their goals;
- *social ability:* intelligent agents can interact with other agents.

When we are talking about agents in the remainder of this thesis, we mean that type of agent equipped with the characteristics of an intelligent agent.

Agents and Objects

In [3], the authors present a comparison between agents and objects. In this section, we briefly summarize the three main differences between them for a better understanding of what agents are.

First, agents are more autonomous than objects per definition. This means that an agent can decide on its own whether or not to perform an action on request from another agent. In contrast, an object has by definition no control whether or not one of its public methods is executed after it is called by some other object's method. Second, agents act by definition reactive, proactive, and social. Such types of behavior are missing in the description of the standard object model. Third, in an multiagent system each agent is assumed to have its own thread of control.

2.1.2. Architectures for Intelligent Agents

In this section, we describe architectures for the following two classes of agents based on [3].

First, we consider *reactive agents* in which the decision of the agent's next action depends on the situation in which the agent is currently situated in. Second, we consider *belief-desire-intention (BDI) agents* in which decision making is based on the current state of data structures representing the agent's beliefs, desires, and intentions.

Reactive Architectures

The main idea of this architecture is, that intelligent behavior is a product of the interaction between agents and the environment as well as that intelligent behavior is a result of the interaction of different simpler behaviors.

Such a behavior can be implemented as rules of the following form.

$$situation \rightarrow action$$

This rule simply maps the state of the environment as input data directly to an *action* that can change the state of the (local) environment. Furthermore, it should be noted that many behaviors can be executed simultaneously. This architecture can be implemented as a hierarchical state machine as described in [3]. The lower a behavior is in the hierarchy, the higher is its priority. Therefore, lower behaviors are able to prevent the execution of higher behaviors in the hierarchy.

Belief-Desire-Intention Architectures

The BDI architectures are based on practical reasoning as described in detail in [3]. This means for an agent that it decides round by round which action it performs to reach its goals. The two main processes are to decide *what* the agent's goals are and *how* the agent is going to achieve them. In the following, we explain what is meant by the terms beliefs, desires, and intentions.

- *Beliefs:* The information about the agent's environment. The beliefs will be recomputed within a given interval based on the agent's perceptual input and the current beliefs.
- *Desires:* The current options of an agent. Which options are available depends on the current beliefs and intentions.
- *Intentions:* This set represents the agent's current focus. Based on the current intentions, an agent selects the next action to execute.

The entire reasoning process will be updated continuously within a given time interval and intentions are based on the previously-held intentions as well as on the current beliefs and desires.

2.1.3. Fields of Application

The field of applications, where multiagent systems are applied, is multidisciplinary in nature. Examples of related disciplines are cognitive psychology, sociology, organization science, economics, philosophy, and medicine [3].

A concrete example for a cooperative multiagent application is the *RoboCup* [15] robot soccer domain. In *RoboCup*, there are several different leagues. The author of this thesis was member from 2010 to 2011 of the team *B-Human* [16] with focus on the behavior, especially roles and tactics. *B-Human* is one of the most successful teams of the *Standard Platform League*. In this league, all teams have to use the same hardware. For this reason, teams have to focus on the software development for their autonomous robots and do not have to build their own robots. Since 2008 the *NAO* [17] humanoid robot is used in the *Standard Platform League*. Teams play against each other in national and international competitions.

A team in the *Standard Platform League* consisted of four autonomous robots in 2011 [13]. One of the robots is depicted in Figure 2.1. The behavior control of a robot is described as a hierarchy of state machines. The decision of the robot's next action is based on the current state of the robot as well as on input data, for example, sensor input or communication input. Robots of one team are able to communicate with each other. As the robots are autonomous, they do not get any input from any human with exception of the referee during the game. More information about the current state of the team *B-Human* can be found on their homepage [14].

The aim of the *RoboCup* is to solve difficult real-world problems with the knowledge gathered from robots playing soccer.

2.1.4. Tools for Agent-Based Modeling and Simulation

As starting point of our simulation work we examined different tools for ABMS to figure out which one is suitable for our purpose. We concentrated on the following three open source applications: NetLogo [18], Gama [19], and Repast Simphony [10]. A more detailed overview about available ABMS tools can be found in [20]. The first tool we considered as unsuitable is NetLogo, since the other ones provide a richer set of features. These are, for example, charts of desired properties at runtime, support of networks/graphs, or the evaluation of the simulated data.

Finally, we decided for Repast Simphony as simulation framework. There are mainly two reasons for this decision. First, the ability to use Java as programming language to build models. Second, that Repast Symphony is maintained over more than ten years. The first reason means that we can use Plain Old Java Objects (POJOs) to describe agents. Therefore, each project member who is familiar with Java can understand and manipulate the model for their own experiments without learning a new programming language.

Repast Simphony

In this section, we will briefly describe the key features that are provided by the Java ABMS framework Repast Simphony [10, 21], which comes as an eclipse [22] plug-in. This framework provides a Graphical User Interface (GUI) to control the simulation at runtime. This means, we can start the simulation with a selected set of parameters as well as stop, and continue the simulation. Furthermore, Repast Simphony provides time series or histogram charts of desired properties at runtime and the evaluation of the simulated data with tools like *R* [23] or *Weka* [24].

As described in [21], an ABM contains the following three elements.

1. A set of agents with their attributes and behaviors.
2. Relationships between the agents and possibilities to interact with other agents.
3. The environment in which the agents live in and interact with.

Repast Simphony supports three different ways to model agents. Firstly, one can use the GUI to create agents graphically using state charts. Secondly, one can use ReLogo, a integrated language based on Logo [25], to create the ABM. Thirdly, one can use Java and model agents as POJOs. We decided to work with POJOs for all ABMs presented in this thesis.

The main tasks of modeling agent interactions are the specification of agent relationships and the dynamics which rules the mechanism of the interactions. To model relationships between agents, Repast Simphony provides the following topologies [21].

1. *Soup.* An unordered structure in which agents do not have locational attributes.
2. *Grid.* The location of an agent is determined by its position in a grid. The neighborhood of an agent is represented by cells surrounding it.
3. *Euclidean Space.* Agents live in 2D or 3D spaces.
4. *Geographic Information System (GIS).* Agents live in realistic geo-spatial landscapes.

5. *Networks.* Edges of a network can link different types of agents (vertices). One simulation can contain several networks representing different semantics. Repasts network library provides some methods to retrieve related agents and to add agents to a network.

The main object of a ABM is the *context.* It initializes a simulation run at start-up and contains all instantiated agents and projections. Each agent has to be assigned to a context and one agent can be contained in any number of projections.

To execute the agents behavior, Repast Simphony provides an own system clock. This means, that at each tick an agent can execute desired actions. Whether an agent executes an action and also which action will be executed, depends on the internal state of the agent and on the local environment.

Actions or methods can also be scheduled to occur at desired time (system tick). Furthermore, methods can be scheduled using the *watch* mechanism. An agent monitors state changes of other agents in the neighborhood and executes its own behavior as a result of these changes. This mechanism enables a kind of communication between agents in a defined local neighborhood.

In summary, Repast Simphony is the most complete ABMS framework based on Java [20] providing features like a representable system-state at runtime, an own system clock and scheduling, genetic algorithms, neuronal networks, regression, and batch-runs with different parameter ranges. However, Repast Simphony does not support the representation of an individual agent at runtime and for communication purposes only the watch mechanism is available. As far as communication is concerned, there is an approach presented in [26] which combines Repast Simphony with the JAVA Agent DEvelopment (JADE) [27] framework. Thus, more communication possibilities are available if required.

2.2. Software Evolution

The field of software evolution is nowadays a well-known research area in software engineering [28, 29]. The pioneer of this research area was Manny Lehmann, who examined limitations of the classical view of software engineering.

This classical view is dominated by the waterfall model for software development proposed by Royce in 1970 [30]. This model consists of the following phases for the life-cycle of a software system: requirements, design, implementation, verification, and maintenance. In this context, maintenance represents the last phase after the software is delivered. Furthermore, it is assumed that requirements no longer change a lot after the delivery of the software and that maintenance consists only of bug fixes and small changes. According to the IEEE 1219 Standard for Software Maintenance [31], maintenance is defined as:

> "the modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment".

The limitations of this process model for software systems are mainly based on the strict and rigid definition of the different phases and the fact that the entire requirements are occasionally known at the starting point of a software project.

With this limitations in mind, Lehman analyzed the change process of the IBM operating system OS/360 [32, 33] and started to formulate his laws of software evolution in the seventies. His early results were confirmed in later studies [34] analyzing other software projects. Lehman used the term E-type software, describing systems that must be evolved because they "operate in or address a problem or activity of the real world". This means, that such a system has to be adapted to the real world during its lifetime. Lehmans laws of software evolution are presented in Table 2.2.

The following definition of the term software evolution by Lehman et al. can be found in [36]. There is said that Software evolution means

> "the consequence of an intrinsic need for continuing maintenance and further development of software embedded in real world domains".

As mentioned at the beginning of this section, software evolution is nowadays a prominent research field in software engineering. Today one can use software evolution and software maintenance as synonyms and maintenance is part of the pre-delivery as well as the post-delivery phases [28]. Some evolution-related research topics are, for example, software quality, software measurement, configuration management, reverse engineering, and testing. Main entities to analyze, in order to get a better understanding of the evolution of a software projects, are people (e.g. developers, tester), artifacts (e.g. files, classes, methods), and bugs.

For the simulation of software processes we are primary interested in information which represents the state of the structure as well as the quality of the software evolving over time. Another important aspect to analyze is the activity of developers contributing to the software project, because their changes to the software are responsible for state changes of the software. This information must be available in the data sources used for mining processes described in Section 2.5. The following section explains the most important data structures and measurements used in this thesis in order to describe the evolutionary process of software projects.

2.2.1. Software Metrics

If you want to know something concrete about a software project, the software itself, or the quality of a software, you have to measure it somehow. Also in the field of

No.	Name	Law
I (1974)	Continuing Change	E-type systems must be continually adapted otherwise they become progressively less satisfactory.
II (1974)	Increasing Complexity	As an E-type system evolves its complexity increases unless work is done to maintain or reduce it.
III (1974)	Self Regulation	The E-type system evolution process is self regulating with a distribution of product and process measures close to normal.
IV (1980)	Conservation of Organizational Stability	The average effective global activity rate in an evolving E-type system is invariant over product lifetime.
V (1980)	Conservation of Familiarity	As an E-type system evolves all associated with it, e.g., developers, sales personnel, users must maintain mastery of its content and behavior to achieve satisfactory evolution. Excessive growth diminishes that mastery. Hence the average incremental growth remains invariant as the system evolves.
VI (1980)	Continuing Growth	The functional content of E-type systems must be continually increased to maintain user satisfaction over their lifetime.
VII (1996)	Declining Quality	The quality of E-type systems will appear to be declining unless they are rigorously maintained and adapted to operational environment changes.
VIII (1996)	Feedback System	E-type evolution processes constitute multi-level, multi-loop, multi-agent feedback systems and must be treated as such to achieve significant improvement over any reasonable base.

Table 2.2.: Lehmans laws of software evolution (adapted from [35]).

software engineering, the famous quote from Sir William Thomson, First Baron Kelvin from 1883 [37] is still applicable

> "When you can measure what you are speaking about,
> and express it in numbers, you know something about it".

A quantified statement about a product or a software process is called *metric* [38]. In this case the measure in the actual sense and not not in the mathematical meaning is meant. In the IEEE Std 610.12 [39] metrics are defined as follows.

metric: "A quantitative measure of the degree to which a system, component, or process possesses a given attribute. See also: quality metric."

quality metric: "(1) A quantitative measure of the degree to which an item possesses a given quality attribute.
(2) A function whose inputs are software data and whose output is a single numerical value that can be interpreted as the degree to which the software possesses a given quality attribute."

The *what* to measure and the *how* to measure play an important role answering the questions about the software under investigation. The question what metrics are relevant is more difficult than it may seem at first. To answer it Basili et al. presented the prominent *Goal Question Metric* [40] approach. Thereby, questions about the software are created based on defined goals and software metrics regarding the software quality, the software process, or the software product are used to answer them. A framework for understanding and using measurement as well as metric foundations are described in [41]. It helps to choose a suitable measurement.

Metrics could be sorted by the area of application. The following areas are based on [38]. As examples we use metrics which are considered for the simulation of software evolution.

- *Cost metrics* concern cost, personnel requirements, and development time of a project. For simulation purposes we are interested in the number of developers contributing to a project over a certain period of time.
- *Bug metrics* represent bug information such as the number of open, closed, and re-opened bugs which are important for simulation purposes.
- *Volume metrics* include all information regarding the size. For the simulation of software evolution we require the size of the entire project (number of files) as well as the size of individual files (lines of code).
- *Quality metrics* give statements about a certain quality aspect of the software. For our simulation model, we are mainly interested in complexity and maintenance aspects.

Metric	Type	Name
LOC	Size	Lines of Code
McCC	Complexity	McCabe's Cyclomatic Complexity
WMC	Complexity	Weighted Methods per Class
NOI	Coupling	Number of Outgoing Invocations
NII	Coupling	Number of Incoming Invocations

Table 2.3.: Overview of used software metrics.

The used software metrics in this thesis are presented in Table 2.3. The metric Lines of Code (LOC) counts the lines of code of a method or class including empty and comment lines. The McCabe's Cyclomatic Complexity (McCC) describes the complexity of a method based on the number of independent control flow paths [38]. On class level, the metric Weighted Methods per Class (WMC) calculates the complexity of a class by summing up the methods McCC of the class. The coupling is described by the metrics Number of Outgoing Invocations (NOI) and Number of Incoming Invocations (NII). NOI counts the number outgoing method calls and NII counts the number of incoming method calls.

Furthermore, for object oriented programming languages exist specialized metrics. The best known have been introduced by Chidamber and Kemerer [42]. These are, for example, the number of methods per class, the depth of inheritance tree of of a class, and the coupling between object classes. These metrics are important for the simulation of software refactorings where an abstract software graph evolves over time. In this scenario, the manipulation of this graph induces an update of object oriented metrics as well.

2.2.2. Change Coupling Graph

The change coupling graph is a undirected graph with a set of nodes representing the files of the software and a set of weighted edges representing the coupling between files. According to Ball et al. [2] an edge is created between files that are changed several times together in one commit. If an edge already exists, then the weight of this edge increases. The authors of [2] showed that files, that are often changed together in one commit, are semantically related. Because of this semantic relationship we use this kind of graph to represent the simulated software.

It is easy to imagine how such a graph changes over time due to the developers' work. Required metrics are the number of developers and the size of the project. These metrics and the way the developers work can be retrieved from software projects by mining relevant data sources.

2.2.3. Abstract Syntax Tree

An abstract syntax tree (AST) represents the structure of source code in a more abstract way than the compiler parse tree does. The nodes of the tree represent constructs of the source code. Like the change coupling graph, this tree changes over time due to the developers' work. For modeling and simulating refactorings, we extend the simulation model with entities representing classes and methods. Therefore, we require the AST enriched with metrics for the size and complexity of classes and methods. This information can be gathered from software projects by mining.

2.3. Refactoring

To perform a software *refactoring* means to improve the design of the code after it has been written [6]. The following definition is based on [43]:

> "*Refactoring* is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior."

When an existing software system will be updated over time, the structure of the code according to the initial design gets worse. Using refactorings one can rework it into well-designed code. This can be done in several small and simple steps. The benefits of refactoring are that the code is more readable to current or future developers and that the maintainability increases.

Below we will describe three of these steps which are used in this thesis, namely the refactorings *move method*, *extract method*, and *inline method*. Definitions are based on [6].

- *Move method* will be applied if a method calls more methods or features of another class than from its own. To resolve this high coupling the method will be moved to the class with the most calls and all affected references will be adapted. The refactoring can also be applied when classes have too much behavior.

- *Extract Method* will be applied to large methods or if code fragments can be grouped together. It creates a new method that is called from the old one and moves code from the old method to the new one. In other words, the original method has been split. Short and well-named methods have two advantages. First, small and finely grained methods are easier to use by other methods. Second, higher-level methods are more readable.

- *Inline method* is the opposite of Extract Method. If, for example, the body of a method is as clear as the name one can inline it. To inline a method one has to find all calls of a method and replace them with the body of the method. Afterwards, the method definition can be removed.

Tools

There are only a few tools available which can find applied refactorings in software projects. We investigated the following two of them.

- *Ref-Finder* [44]: This tool can identify refactorings between two program versions and implements sixty three refactorings of Fowler's catalog [6]. Ref-Finder uses logic programming to infer concrete refactoring instances. Therefore, each supported refactoring type is expressed as in terms of template logic rules.

- *RefDiff* [45]: This tool can identify refactorings in the commit history of git repositories. RefDiff supports 13 prominent refactoring types of Fowler's catalog [6] and uses heuristics based on static code analysis as well as code similarity metrics to identify refactorings between two program versions.

Which tool we use to find refactorings between two code versions and where the differences between both tools are will be discussed later on in Section 5.3.2 and in Section 6.3.

2.4. Graph Transformations

Graphs are well known in computer science as well as the theory of rule-based graph transformations [46–48]. Some prominent examples for graphs are diagrams of the Unified Modeling Language (UML) [49] representing the abstract syntax of a program, entity relationship diagrams [50], or the AST of a program describing the source code. In general, graphs are used to describe relationships, represented as edges, between objects, represented as vertices. Furthermore, graphs can be dynamic or static. For example, a UML class diagram represents a static view of the software whereas a software graph (e.g. the AST) of a program under simulation is dynamic during the execution. The theory of graph transformations, which will be introduced below, provides the possibility to transform the structure of graphs rule-based.

The most fundamental definitions that are required to understand the theory of the rule-based graph transformations used in this thesis are based on [51].

2.4.1. Definitions

In this section, we introduce the terms required to understand rule-based graph transformations. We focus on directed, edge-labeled graphs in combination with rule application following the so-called double-pushout approach (DPO) [51, 52].

Graph

A multiple directed, edge-labeled graph G over Σ is a system $G = (V, E, s, t, l)$. Then let Σ be a given set of labels. In this definition, V is a finite set of vertices, E is a finite set of edges, a *source* $s(e)$ and a *target* $t(e)$ are assigned to every edge $e \in E$ with the mappings $s, t : E \to V$, and the mapping $l : E \to \Sigma$ assigns a label to every edge in E. An edge $e \in E$ with the same node as source and target $s(e) = t(e)$ is called a *loop*. The components of G can also be written as V_G, E_G, s_G, t_G, and l_G, respectively. The set of all graphs over the set of labels is denoted by $\mathcal{G}_\Sigma$.

The notion of this graph provides enough flexibility to cover other types of graphs. We assume that we have to deal with dynamic graphs that serves as inputs for algorithms or processes (e.g. the simulated software graph). Thus, we introduce rule-based graph transformations to define rules for well-structured graph manipulations.

Subgraph

A subgraph of a given graph G is represented by a subset of vertices and edges and every edge of the subgraph has the same source and target node and the same label as in G. More formally, let $G \in \mathcal{G}_\Sigma$ be a subgraph of the graph $H \in \mathcal{G}_\Sigma$. This is denoted by $G \subseteq H$, if $V_G \subseteq V_H$, $E_G \subseteq E_H$, $s_G(e) = s_H(e)$, $t_G(e) = t_H(e)$, and $l_G(e) = l_H(e)$ for all $e \in E_G$.

One can obtain a subgraph by removing some nodes and edges. After removing a node, it is required to remove all incident edges. This is called *contact condition*.

Graph Morphism

For two graphs $G, H \in \mathcal{G}_\Sigma$ a graph morphism $g : G \to H$ is a pair of structure-preserving mappings $g_V : V_G \to V_H$ and $g_E : E_G \to E_H$. The image of G in H is called a *match* of G in H. Furthermore, the match of G regarding the morphism g is the subgraph $g(G) \subseteq H$, induced by the pair of mappings $(g(V), g(E))$. Due to the structure-preserving nature of g the contact condition of subgraphs is valid.

2.4.2. Graph Analysis

In order to be able to make statements about the quality of the simulated software, the real and the simulated software graph are compared. Graph theory is a well-researched area (see [53–55]) from which we use only a fraction for our analysis. For the purpose of analysis we use Gephi [56], a visualization and exploration software for all kinds of graphs and networks, or R [23]. The following graph metrics provided by Gephi are used for this comparison.

Degree: The degree of a vertex is the number of vertexes incident to it.

Weighted Degree: If considering a weighted graph, the weighted degree of a vertex is the sum of weights of the vertexes incident to it.

Density: The density of a graph represents how close the number of edges is to the number of maximum edges of the graph.

Modularity: The modularity of a graph represents how good a graph can be divided into highly connected areas, for example, clusters.

Diameter: The diameter of a graph is the maximal shortest path between any two vertices.

For the analysis of software graphs, we are also interested in the subdivision of the graph in cluster. These strongly interconnected structures represent semantically related parts of the software [2]. For example, they can represent different components such as GUI or database. A cluster is often named a *community* in literature [57].

To find clusters in a graph there are several algorithms available and a comparison can be found in [58]. Gephi uses for this purpose an approach based on the modularity proposed by Blondel et al. [59].

2.4.3. Rule-Based Graph Transformation

Graph transformations are used to apply local changes based on rules to graphs. A rule describes which part of a graph has to be replaced by some other graph [51, 60].

A rule $r : L \Rightarrow R$ consists of a left-hand side L and a right-hand side R, both are graphs. The starting point in G of the rule is represented by L and the effect of the rule application is described by R.

To apply a rule r to a graph $G = (V, E, s, t, l)$ one has to execute the following three steps, which finally lead to the derived graph H. A match is given as the morphism $g : L \cup R \rightarrow G \cup H$ with $g(L) \subseteq G$ and $g(R) \subseteq H$.

1. Find a match of L in G.

2. All vertices and edges that are matched by $L \setminus R$ are deleted from G which results in the intermediate graph Z. In this step we must make sure that the result of $Z = G \setminus g(L \setminus R)$ is a valid graph. This means that no dangling edges, caused by removed target or source vertices, remain after this step.

3. The graph H is created by gluing Z with $R \setminus L$, this means $H = Z \cup (R \setminus L)$.

To restrict the allowed graph transformations one can use a *type graph*. It is similar to an UML class diagram and expresses which nodes can be linked with a certain edge type [61].

Using the DPO [52], there are no dangling edges in the new created graph H after the application of a transformation rule. In contrast to this approach, the single-pushout approach (SPO) [62] performs only one graph derivation without the intermediate graph in the middle. The SPO is more powerful without the restriction of the gluing condition, but the graph could be destroyed by the transformation – edges without source or target nodes could exist after the rewriting step.

2.5. Mining Software Repositories

Since various tools for data storage and communication are used for organizing and configuring software projects, it is possible to get information about the project by analyzing the data stored by the tools. With this data available, especially with increasing data of large Open Source Software (OSS) communities, Mining Software Repositories (MSR) has become a popular field of research over the last few years. An overview of the wide range of research and application areas is published in [63].

To simulate the evolution of software processes, we are interested in the software changes, their causes, and their impact [64]. To get the required information, we have to analyze the source code of the software. Analyzing the source code means, that we can retrieve desired software metrics (see Section 2.2.1) of each version of the software and that we can compare these metrics with the metrics of other software versions. Based on this, we can describe trends and patterns that represent the evolution of the software. This information serves as input for our simulation model. Specifically, these are, for example, the size of the project, the size and complexity of different software entities, the number of developers contributing to the project, and the effort spent by the developers.

Because we want to examine the quality of the software, we are also interested in the number of open, re-opened, and closed bugs. These information are stored in Issue Tracking Systems (ITSs). There are different data sources available for analysis.

Common problems occurring during the mining process are, for example, the linkage between different entities which could be stored in different data sources like files and bugs. A lot of research has already been done on this topic, for example, in [65–68].

Another common problem in mining software repositories is to identify the identities (e.g., logins or e-mail addresses) of developers in software repositories or other data sources that represent the same physical person. To determine, for example, the effort one person spent to the project one has to merge all identities representing this person. An overview of different identity merge algorithms is given in [69].

Furthermore, the tools used to find software refactorings as described in Section 2.3 make also use of mining techniques to find occurring refactorings between two different code versions. For this differencing task, the AST is used to analyze fine grained information about changed software entities like classes or methods.

Data Sources

As mentioned before, software projects are often organized in the way that project-related data is managed in different data sources. The source code is stored in Version Control Systems (VCSs), bugs are managed in ITSs, and for project related communication Mailing Lists (MLs) are used. Furthermore, even social media like Twitter can be used for communication purposes.

In the following, we will briefly describe the most popular data sources before we introduce mining frameworks that gather information from all available data sources of a project and provide one interface for queries.

A VCS stores every version of a software document (e.g., source code file or documentation file) in a database. In practice, only a delta is saved when a file is changed in a commit. In addition to the changed files, a commit contains the author, the commit date, and a commit message. Therefore, such a repository contains the entire history of a software project. We distinguish between a centralized VCS and a distributed VCS. The first one has only one central repository on a central server and each client can checkout a working copy from there. Prominent examples for centralized VCSs are Subversion [70] and the Microsoft Azure DevOps Server [71]. The distributed VCS is not limited to one central repository and each client checkout contains a working copy as well as the whole repository. This reduces the risk of data loss if the central server crashes. Well known examples for distributed VCSs are git [72] and Mercurial [73]. For mining purposes, distributed repositories have the advantage that all data is available on the local system [74]. This means, that after the repository checkout the entire history is analyzable without additional effort or network traffic.

Another important data source is the ITS. This system stores and manages all project related issues in a database. Developers, testers and users can create tickets in the ITS concerning bugs, desired improvements, or feature requests. Each ticket contains at least the following attributes: id, severity, priority, status, date of creation, creator, and description. The status gets from *new* after the creation of the ticket over *resolved* after some maintenance work to *closed* after confirmation of the fix by quality assurance. If the issue remains, the ticket could be *re-opened* for further improvement. The significance of the ticket is represented by the severity. Typical severities are blocker, critical, major, minor, or enhancement. Well-known examples for ITSs are Bugzilla [75] and Jira [76].

If MLs are used for the project communication, they can provide valuable information about developer activities, states and behavior.

For mining purposes in this thesis, only data stored in VCSs and ITSs are analyzed to retrieve the required simulation parameters.

2.5.0.1. Mining Frameworks

Each of the above mentioned data sources has an own database and infrastructure. When you are interested in data of different data sources you have to deal with different interfaces. For specific questions, there are several tools for data extraction available and most of them store the gathered data in own databases and therefore provide its own interface. For example, when you are interested in static source code analysis you can use the tool SourceMeter [77] to retrieve source code metrics. According to the documentation the calculated metrics by SourceMeter are divided into six categories, which are the following: Cohesion metrics measure the dimension of cohesion between software entities. Complexity metrics measure the complexity of given software entities (usually algorithms). Coupling metrics measure the amount of interdependencies between software entities. Documentation metrics measure the amount of comments and documentation of software entities. Inheritance metrics measure the different factors of the inheritance hierarchy. Size metrics evaluate the fundamental characteristics of the software analyzed in terms of various cardinalities, for example, lines of code, or the number of classes or methods. SourceMeter stores all extracted information into a database.

In order to facilitate the mining process, mining frameworks have been developed in recent years, for example [78]. These frameworks provide one infrastructure to query against specific research questions.

Another mining framework is SmartSHARK [79] developed at the University of Göttingen within the Institute of Computer Science. This framework contains a set of different tools that collect data from VCSs, ITSs, and MLs. Furthermore, it collects software metrics and AST statistics. All collected data is stored in a MongoDB. On the analytical side, Apache Spark gives SmartSHARK the required efficiency to analyze this quantity of data. Due to the collection of all data in one MongoDB, it is easy to study research questions that depend on different data sources.

The plug-in of SmartSHARK to find refactorings between source code versions in the VCS is implemented as part of this thesis. Furthermore, the framework for automated simulation parameter estimation implemented in this thesis uses SmartSHARK as data source.

3. Related Work

Contents

This thesis benefits from several research areas. First, we use multiagent systems (see Section 2.1) for the simulation of software evolution. A similar simulation approach to our approach is published by Smith et al.[80]. Further different approaches are discussed in Section 3.1. Second, for the simulation parameter estimation we are mining software repositories. In Section 3.2 we discuss areas of this well-known topic of software engineering that are required to retrieve necessary simulation parameters. Third, to improve the structure of the simulated change coupling graph we model software refactorings (see Section 2.3) using graph transformations (see Section 2.4). Refactorings and graph transformations are well-known research topics. The interaction of both research areas is discussed in Section 3.3.

3.1. Simulation of Software Processes

The simulation of software processes to predict selected aspects of the software under simulation is well known in software engineering [81, 82], but the most important recently published approaches use either System Dynamics (SD) or Descrete Event Simulation (DES) instead of ABMS. This is because ABM is a comparatively new research area. In the following section we present one publication that compares ABMS and SD as well as several different ABMS approaches.

A comparison of ABMS and SD is presented by the authors of [83]. Their studies are based on individual characteristics of developers like the experience or the competence. The authors figured out that the configuration of the SD model is much easier, but the results of the ABM are more realistic. The reason for the easier configuration of the SD model is that the ABM requires a more detailed description of the individual developers to model their behavior. Such detailed developer descriptions are also used to parameterize our simulation model.

In [84], the authors present an approach to analyze developer networks using ABMS. These networks represent the developer contribution to several projects hosted on SourceForge[1]. In this model, developers are equipped with the possibility to join, stay in, or leave projects. For simulation purposes the authors use the multiagent framework SWARM [85] and parameterize their model with data retrieved from SourceForge. The usage of empirical data for parameter estimation is similar to our approach, but the simulation covers more a top level view over several projects whereas our model is much more detailed and the simulation provides a project level view.

Another study that uses ABMS to model software evolution is presented in [86]. The work is aimed to support project managers in their planning by simulating possible future software processes. The authors use data from a software department in an industrial context to estimate the simulation parameters. This work differs from other studies in so far that a maturity model is given, the Capability Maturity Model Integration (CMMI [87]). During the creation of the agent-based model, the number of existing software components and the number of available developers is considered based on the design and the development phase. Then, the developers are assigned to certain (multiple) components. The components switch between different states. Finally, the model is validated by comparing the empirical project duration of different projects with the simulated results. This model is a more specific one than our developed model, but the validation idea to use empirical data to compare it with the output of the simulation is similar to our approach.

An approach that uses ABMS and where the behavior of developers is described very detailed is presented in [88]. In this work the developers' decision making process is based on the Personal Software Process (PSP). Thus, this model is more tailored towards a specific project type using the given process model, e.g. extreme programming, than our model.

An ABM for software processes similar to our model is presented by Smith et al. in [80]. In their work, the developers are the active agents and they can perform a random walk on a grid. When a developer reaches a cell containing a software module or a requirement, it can work on it, and when a developer moves outside

[1]https://sourceforge.net/

the grid, it can leave the project with a certain probability. If the developer works on a module depends amongst other things on the complexity of the module. To work means an immediate change of the state of the updated module. The authors can reproduce different aspects of software evolution, for example, the number of complex entities, the number of touches, and distinct patterns for system growth. In our tests, almost all of them need different parameter sets to get realistic results. The model we proposed has the following differences to the one presented by Smith et al.: First, our model is not grid-based and agents do not perform a random walk. In our work, all instantiated agents live in one environment and relationships are represented as networks. Second, our simulation model for system growth analysis requires only parameters for effort and size to simulate projects that have similar growth trends. Furthermore, our model supports several developer roles and each of them has its own contribution behavior.

3.2. Mining Software Repositories

Since platforms such as GitHub[2], SourceForge[3], and Bitbucket[4] become more popular to host and manage open source projects, more and more data is easily available to researchers. Thus, a broad field of research has developed. For example, research topics concerning software evolution are programming languages, different development stages of the software, or the software management process. An overview is given by Mens in [28]. In the following, aspects of mining software repositories required for the estimation of parameters used for the simulation of software evolution are discussed in more detail.

3.2.1. Software Evolution

As mentioned above there are a lot of publications available dealing with the mining of open source software repositories to analyze the evolution of software projects. Which aspects are important for us? At the beginning of the simulation work we only tried to replicate the growth of a software project. Afterwards, we analyzed and simulated dependencies between files, developers, and bugs. Thus, we discuss work related to the project growth, several dependency networks, and bug occurrences below. Of course, the behavior of the developers is of central importance for the simulation and, hence, discussed in Section 3.2.2.

To analyze the growth of a software project, we need a metric to quantify the size of a project. For this, the number of files, modules [89], classes, or methods could be used. The growth trend of software projects is analyzed by Godfrey and Tu [90]. The

[2]https://github.com/
[3]https://sourceforge.net/
[4]https://bitbucket.org/

authors figured out, that most projects follows a sub-linear trend decreasing over the time. The comparison of the growth of open source projects and closed source projects in [91] reveals that both follow a similar growth trend. Furthermore, the authors show that a linear function for all growth concerning measurements (lines of code, number of functions, complexity) could be fitted. In the work of Robles et al. [92, 93] the authors found also linear growth trends for open source software projects as well as super-linear trends. In [94], the authors found some segments of sub-linear growth while analyzing the number of files and the number of folders. One goal of the simulation models developed in this thesis is to reproduce the respective growth trend of analyzed software projects.

Another important factor of our model is the representation of dependencies between the software entities. We use networks for this purpose. Dependency graphs can be, for example, the hierarchy graph representing the inheritance structure of the software, the call graph representing the relationship between classes and functions based on method calls, or the change coupling graph. The latter represents clusters of files that are changed several times together in a commit (see Section 2.2.2). According to [2], files of one cluster are semantically related. Because we are interested in making statements about the quality of the simulated software, we also consider the change coupling graph under this aspect. In [95], the authors presented that hard to maintain parts of the software are related to a high change coupling degree. Concerning other quality aspects like the bug localization or the number of defects the author of [96, 97] also analyzed the evolution of the change coupling graph.

The occurrence and the fixing of bugs in all its subareas are well examined research topics. There are publications available concerning, for example, the linking between bugs and software entities [65], the fixing of bugs [98], the classification of changes: buggy or clean [99], or the prediction of the severity of reported bugs [100]. For simulation purposes it is important to know in average how long a bug is alive. The authors of [101] investigated exactly this question using machine learning methods. A classification of bugs into fast and slowly fixed ones is also part of their work.

3.2.2. Developer Classification and Contribution Behavior

When it comes to developer classification it is a common approach to divide developers into core and peripheral [102–106]. This classification is due to the well-known onion model [107]. The main assumption of this model is that a small amount of developers contribute most to the project. A quantification is given in [108], the authors consider the top 20% of all contributing developers as core developers. The main differences between both developer types are that the core developer is more active and contributes more to the project. Furthermore, in [104] the introduced structural complexity of both developer types is analyzed. The authors figured out, that core developers insert less complexity. For our simulation model the classification is slightly different. We differentiate between core, major, and minor developers.

The complexity is only considered for the simulation of refactorings, for all other simulation models the complexity is omitted. A much more complex role classification based on bug related metrics is presented in [109]. Such a role consideration is not suitable for our simulation purposes.

The investigation of the developer's contribution behavior is a prominent research area of software engineering [110–112]. A uniform definition of the term developer contribution can not be found, but it can be considered as the work a developer spent to the software project [113]. One can use many metrics to quantify the contribution such as the number of commits, files changed per commit, or lines of code per commit. We are using all of these three metrics to parameterize our simulation model for software evolution.

Based on the contribution behavior of the software developers the authors of [114] defined the ownership of a file. The owner of the file is the developer who edited the highest percentage of it. The owner can change if another developer invests more work than the original owner. Thus, the creator of a file is not automatically its owner. For simulation purposes, we have a slightly different definition of the ownership. Instead of counting the edited part of a file we use the number of touches as characteristic feature for the owner of a file.

A model-based mining approach to reveal the developers (contribution) behavior is presented by Makedonski [115]. Due to the model-based approach, it is possible to perform mining for a variety of software engineering relevant tasks by adapting desired models. A possible application scenario is the prediction of bugs, based on a deep analysis of causes and impacts of software changes.

3.2.3. Commit Analysis and Source Code Differencing

For more detailed simulation models we require more information about the difference between two different source code versions. Desired information are, for example, the size and complexity changes of classes and methods from version to version. Furthermore, we require more information about the commit type. A commit type could be, for example, a refactoring or a bugfix.

An approach to classify commits is given by Hattori and Lanza [116]. They classify the commits into four major activities: forward engineering as a development activity; and reengineering, corrective engineering and management as maintenance activities. The classification is a keyword-based analysis of the commit message. Example keywords for a forward engineering commit are *implement*, *add*, and *new*. A corrective engineering commit could be classified, for example, with a commit message including the words *bug*, *fix*, or *error*. Furthermore, the commits are also divided into four size classes. For example, tiny commits contains 1 to 5 changed files. By doing so, the authors figured out that 80% of the corrective engineering commits are tiny ones. For commit classification purposes, we are using a similar approach with a slightly different keyword list.

To find refactorings between two source code versions there are a lot of publications available[117–119], besides the both earlier mentioned tools [44, 45] described in Section 2.3.

Tsantalis et al. [120] developed RMiner, a tool to detect refactorings between two software revisions. RMiner can detect 15 prominent refactoring types of fowlers catalog [6] using an AST-based statement matching algorithm that does not require user-specific code similarity thresholds. RMiner only analyzes files that are added, deleted, or updated between two revisions. For validation purposes, the authors created an oracle consisting of 3188 refactorings found in 538 commits from 185 open source projects with the help of several tools and experts. According to the evaluation against this model the authors stated that RMiner is a significant improvement over tools like RefDiff [45] and it achieved 98% precision and 87% recall. Furthermore, RMiner is more efficient than other available tools. Since RMiner was not available when the mining of refactorings used in this thesis was implemented, our approach is based on RefDiff [45].

To find parameter for change patterns, such as a metrics, in source code changes we have to analyze the evolution of source code files in detail. This means, we need to analyze the changes of two software versions on AST level. There are well-known algorithms available [121, 122] that deal with that problem. When we know *what* parts of the software have changed in one commit, we can figure out *how* this software parts changed by performing a static source code analysis of both versions using tools like CVSAnalY [123, 124] or SourceMeter [77]. Some AST related metrics are also stored in the SmartSHARK [79] database described in Section 2.5.0.1. For the parameter estimation process in this thesis we use SmartSHARK.

A dataset containing fine-grained metrics information is published in [125]. The authors analyzed 7 open source projects using RefFinder [44] and extracted more than 50 types of source code metrics at class and method level for 37 releases. All considered projects are implemented using the Java programming language and hosted on GitHub. Based on this dataset the authors figured out that classes with lower maintainability are subject to more refactorings in practice than classes with higher maintainability. To the same result, but on method level, the authors come in [126]. Furthermore, the authors figured out that the application of refactorings decrease size, coupling, and clone metrics. Such a published data set is valuable for scientists who want to deal with the analysis of software refactorings without having to worry about the sometimes costly mining work. For our purposes this dataset is not fine-grained enough, because we need the metrics at the commit level instead of the release level.

3.3. Modeling Refactorings using Graph Transformations

To enrich the simulated change coupling graph, we simulate software refactorings. For this, we use graph transformations. A transformation rule represents a metrics and thus it can be applied to the software graph. We discuss publications related to the modeling of refactorings using graph transformations below.

Graph rewriting systems to describe program transformations are introduced in [127–129]. This work forms the basis of formalizing refactorings using rule-based graph transformations. In these papers, an own graph representation is introduced and the representation of the source code as AST is not used.

The feasibility of using graph rewriting systems for specifying refactorings is shown in [61]. In this paper, also an own graph representation for programs is introduced. Based on this the authors show how refactorings can be expressed by graph productions.

In [60], the authors present an approach to maintain consistency between code and model diagrams when a refactoring is applied. To model refactorings rule-based graph transformations are used. The model is represented as UML diagram and the code is represented as AST. To handle both different graphs in a separate but consistent way, the authors use concepts of distributed graph transformation [130]. Furthermore, the usage of transformation units [131] enforces the synchronization of the transformations in both diagrams.

The work described above presents theoretical models that are detailed enough so that they can be used as rules to be implemented in an Integrated Development Environment (IDE) to execute refactorings on arbitrary object oriented source code. For our simulation models, we do not need that level of detail. Therefore, the models developed in this thesis are much more abstract than the models described in the publications above.

4. Evolution of Agent-Based Simulation Models

Contents

This chapter describes the evolution of developed ABMs to simulate different aspects of software processes. All developed models are improved step by step including lessons learned by predecessor models. This model evolution and adaption process represents a key part of the research work established for this thesis. First, we present a grid-based model adapted from Smith et al. [80] and discuss limitations given by the grid-based approach. Then, we briefly describe a growth model where agents do not have any modeled dependencies. Later on, we introduce models containing network based dependencies between the agents. Furthermore, we present a model to simulate software refactorings based on graph transformation rules. Finally, some implementation details to instantiate the model are presented.

As mentioned in Section 2.1.4, we are using the ABMS-framework Repast Simphony [10, 132] for all modeling and simulation tasks in this thesis.

4.1. Grid-Based Model

The first model for the simulation of software processes that we have created is based on the work of Smith et al. [80]. They analyzed the relation of size, complexity, and effort during the evolution of OSS. Furthermore, the complexity of a software module is considered as a limiting factor in productivity. We implemented most features of this model in Repast Simphony due to better understanding of the tool and ABMS in general with discussed deviations. The proposed model consists of the agents shown

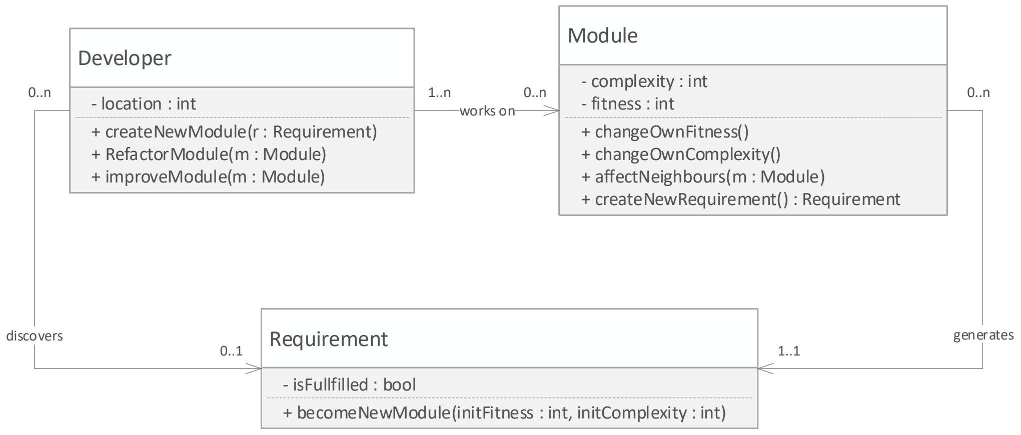

Developer
- location : int
+ createNewModule(r : Requirement)
+ RefactorModule(m : Module)
+ improveModule(m : Module)
0..n
1..n
works on
0..n
Module
- complexity : int
- fitness : int
+ changeOwnFitness()
+ changeOwnComplexity()
+ affectNeighbours(m : Module)
+ createNewRequirement() : Requirement
0..n
discovers
0..1
Requirement
- isFullfilled : bool
+ becomeNewModule(initFitness : int, initComplexity : int)
1..1
generates

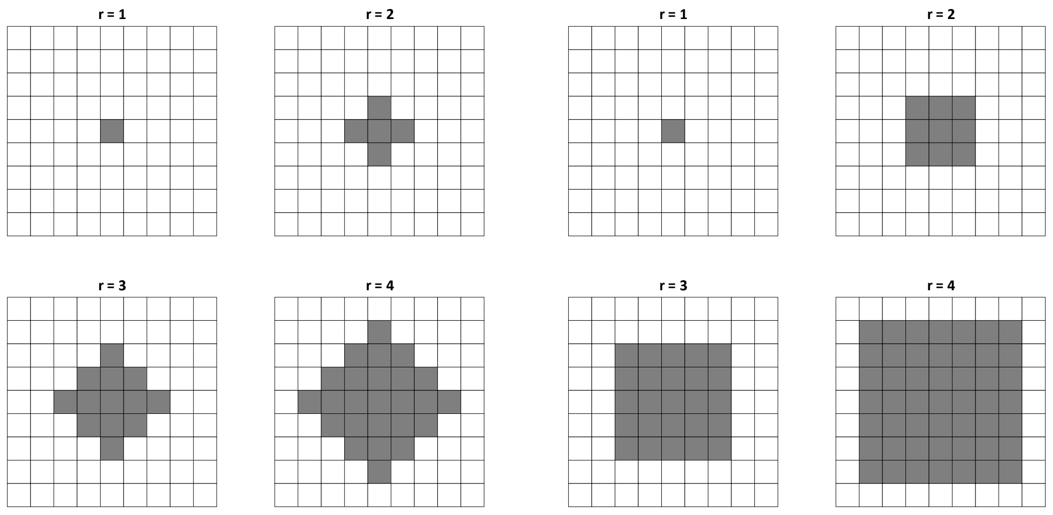
Figure 4.1.: Agent-based simulation model.

in Figure 4.1. According to our implementation, the developers and the modules are considered as *active* agents, which means that they have their own behavior. The requirements are considered as *passive* agents, because they have no own behavior and their state can only be changed from outside by other active agents.

Since this model is grid-based, all agents live and act on a grid with fixed size. Implementing the model with Repast Simphony allows us to use two different neighborhoods. First, we can use the Von Neumann neighborhood [133] containing one central cell and its four adjacent cells for a distance of one as depicted in Figure 4.2a (for distances from 1 to 4). Second, the Moore neighborhood [134] is provided. It contains one central cell and all its eight surrounding cells for a distance of one as depicted in Figure 4.2b (for distances from 1 to 4).

Moving on such a grid, the active agents have the following characteristics to describe the evolutionary process of the software.

Modules are representing arbitrary software entities. For example, a file or a class. A module has two properties as defined in [80] as follows: the fitness representing how good a module fits the requirements and the complexity. The higher



(a) Von Neumann neighborhood of a grid. (b) Moore neighborhood of a grid

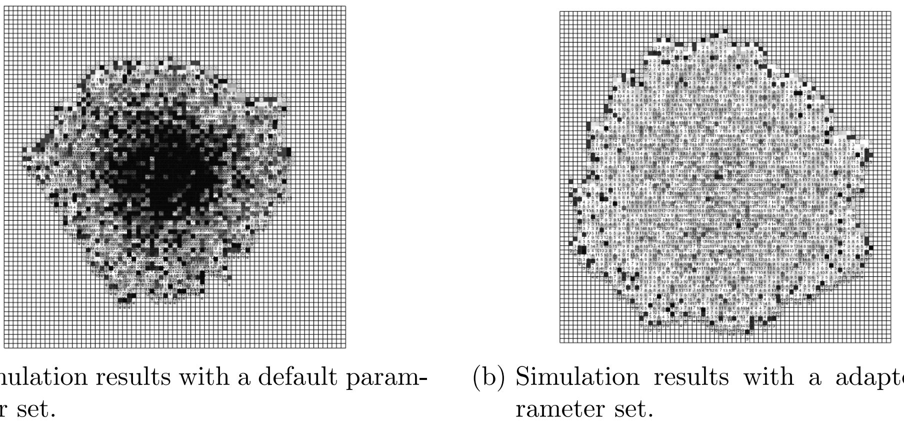
Figure 4.2.: Possible neighborhoods of a grid space provided by Repast Simphony.

the complexity, the less likely it is for a developer to improve the module. The own behavior of a module partly describes the intrinsic aging. Assuming that the fitness suffers from *bitrot* (decrease of fitness over time) with aging while the complexity increases, the module can change its own state according to a given bitrot parameter. Furthermore, a module has a random probability to create a new requirement in an empty cell in its neighborhood. Finally, a module can lower the fitness of adjacent modules after a developer has worked on it.

Developers perform a random walk on the grid. If a developer reaches the border it can leave the project with a certain probability. Depending on the content of the target cell a developer can perform three different actions. First, it can create a new module with initial low fitness and complexity if it is on an unfilled requirement. Second, if a developer is on a module with high complexity and high fitness it can apply a refactoring. This will reduce the complexity of the module by a random amount. Third, being on a module and no refactoring was applied, a developer can work on it, if the complexity is not too high. This work will increase the fitness and the complexity of the module by a random amount.

At the beginning of each simulation run there is only one module and a certain amount of developers. Through the behavior of developers and modules described above the project evolves over time. If there are enough modules with low fitness, they can attract developers and new developers will be created.

For the instantiation of the model a lot of parameters have to be set. There are, for example, the initial complexity and fitness of a module, probabilities for new developers and requirements, the impact of a refactoring or development work, or a threshold for the maximum complexity to develop a module. Using a default parameter set provided by the authors of [80] the fitness of the project under simulation using our implementation is quite low as depicted in Figure 4.3a.


(a) Simulation results with a default parameter set.

(b) Simulation results with a adapted parameter set.

Figure 4.3.: Simulation results of the grid based model simulated with Repast Simphony. A cell represents the fitness of a module. The brighter the cell, the higher the fitness of the module.

By adjusting the parameters, the simulated results as shown in Figure 4.3a could be improved. For this result, we decreased the initial module fitness and complexity, the complexity threshold, and the increase of complexity per development step. Furthermore, and we increased the impact of a refactoring.

In contrast to the model proposed by Smith et al. in our simulation model we omitted the developers motivation as factor to leave or join a project according to its fitness, but we have introduced a threshold for the maximum number of developers. This made the results more reproducible. Furthermore, we changed the behavior of the developers so that they do not move randomly in the grid. They move more goal directed and go to cells in the neighborhood where work is expected, for example, due to a lower fitness.

With such a model we can reproduce the number of complex modules, the number of touches, and distinct patterns for system growth of real software projects. However, we require for almost each of them different parameter sets to get realistic results. These results motivated the following suggestions for improvement, which will be considered in our follow-up models, described in the next sections.

- The random number of developers and the condition for leaving the project are not suitable for projects of different sizes. It seems to be promising to use empirical data based on mining software repositories to figure out how many developers contributing to a project under investigation. Furthermore, we will divide developers into different roles or types.

- The random walk on the grid performed by the developers does not seem to be purposeful enough for modeling software evolution realistically. To model a more goal direct behavior we want to consider the intention of each developer per simulation round. For example, intentions may be to commit, or in more detail, to fix a bug, to apply a refactoring, or to implement a new feature.
- The grid is also not suitable for realistically modeling the dependencies between individual software entities. When using a grid, the project starts growing from a startup cell and each module depends on its neighbors. Such a behavior does not allow to model the architecture of the software system accurately. More precisely, for example, core-modules with many dependencies could not be modeled. To make this possible in a nearby way, other possibilities to model dependencies are necessary.

With all these points in mind, we start to develop a complete new model for the simulation of the evolution of software processes presented, in the next section.

4.2. Network-Based Model for Monitoring Software Quality

In this section, we describe the development of an ABM for the simulation of several aspects of software evolution by modeling the dependencies between software entities as networks. This means that the model presented here no longer contains grid-based dependencies. Of course, the learned aspects of the grid-based model described in Section 4.1 are incorporated into the development work. For the instantiation of the following models, we use parameters generated by mining open source repositories.

We start with a basic model for simulating the growth of a project, for which no modeled dependencies are necessary. Afterwards, we present a model to simulate the lifetime of bugs as first quality aspect of the software under simulation. Finally, we explain a detailed model that can be used to simulate several quality aspects of a software project.

4.2.1. Growth Model Depending on Productivity

To model the growth trend of a software project only two agents are required. Firstly, software entities that represent the software system under simulation. Secondly, the developers actively influence the growth of the project with their contribution to the project under simulation. The ABM to simulate the growth of software projects is depicted in Figure 4.4.

As software entities, we consider files due to the fact that we only need the number of files for growth statements according to [89]. A `File` has only a reference to the developer who created it as attribute. Furthermore, a file can be considered as a *passive* agent because its state depends only on the development work of the developers in this model.

The `Developer` is the *active* agent because its development work *adds*, *deletes*, and *updates* files. This contribution makes the project evolve over time and for growth considerations we can omit the updates performed by the developers. A detailed description of the developers contribution model and required parameters can be found in Section 4.2.1.1.

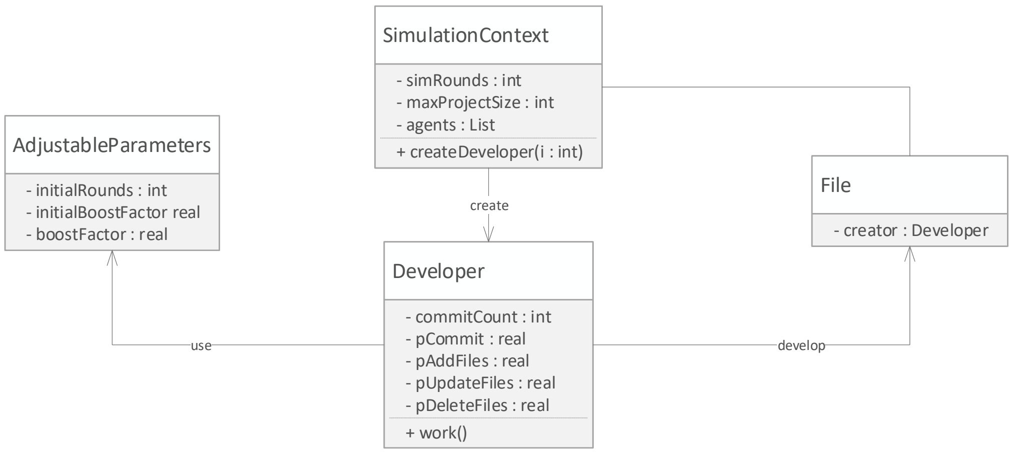
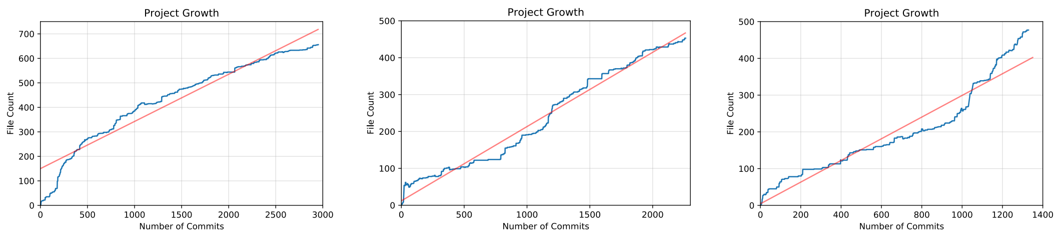


Figure 4.4.: Agent-based simulation model to replicate growth trends of software projects.

The `SimulationContext` is the initial class when executing the simulation model. It creates a given number of developers at the start of the simulation and contains project-specific parameters such as the maximum project size and the number of rounds to simulate. Furthermore, the context knows all instantiated agents, passive as well as active ones, as it is indicated by the list `SimulationContext.agents`. The simulation is round-based and one round in the simulation represents one day in real life. Every turn each developer has the opportunity to work. If the agent works, the project will be further developed through its contribution. This is also the basic principle of all further simulation models. The closer the project size approaches the given maximum size, the fewer new files are created by the work of the developers. This, together with the assumption that stronger growth is expected at the beginning of the project, leads to the typical growth [90] of a software project as depicted in Figure 4.5a. These parameters are gathered by mining open source repositories than described in Section 5.2.

As mentioned in Section 3.2.1, we can have several growth trends. To configure the simulation when mined parameters are not suitable to simulate growth trends like the super-linear one depicted in Figure 4.5c the `AdjustableParameters` can be used. For this purpose, the project is divided into two parts. The first part takes `AdjustableParameters.initialRounds` and the last one takes the rest of the projects runtime. As a responsible parameter for growth, the mined developers contribution can be adjusted for the first part with the parameter `AdjustableParameters.initialBoostFactor` and for the last part with `AdjustableParameters.boostFactor`. These parameters allow us to simulate a more realistic growth trend as with mined average parameters only.

To illustrate different growth trends, we plot the growth in the number of files for the three open source projects commons-io[1], gora[2], and zookeeper[3] and the results are presented in Figure 4.5. According to [91] all growth trends can be fitted with a linear function.



(a) Growth of zookeeper. (b) Growth of commons-io. (c) Growth of gora.

Figure 4.5.: Different growth trends of real software projects.

Since the contribution behavior of the developers are the driving factor for this model, as well as for further models concerning the evolution of software processes, this part will be described in detail in the next section.

4.2.1.1. Developer Contribution Behavior

Due to the round-based nature of the simulation, at the beginning of each round, it must be decided whether a developer is working or not. Working means, that the developer applies a commit. For this, we assume that the commits of one developer are uniformly distributed. This decision depends on the probability `Developer.pCommit`. The number of commits, each instance of the type `Developer` has applied, is stored in the property `Developer.commitCount`.

When a developer applies a commit in the current round, the scope of the work still needs to be determined. Therefore, the number of files that are added and deleted in the commit are required. For this purpose, we assume that the number of file

[1]https://github.com/apache/commons-io
[2]https://github.com/apache/gora
[3]https://github.com/apache/zookeeper

changes follows a geometric distribution. We are using that form of the geometric distribution in which the number of failures until the first success is modeled as shown in Equation (4.1). This means that zero is allowed.

$$P(X = n) = (1 - p)^n \cdot p \qquad \text{for } n = \{0, 1, 2, 3, \dots\} \tag{4.1}$$

For the simulation, this can be interpreted as being a success from a developer's point of view if it has nothing to do and thus zero files have to be updated. All actions a developer can perform are modeled the same way, but for growth trend analysis we only consider delete and add actions while the number of updated files is omitted.

These both properties of a developer, i.e., the probability to apply a commit in the current round and the number of files to add or delete in one commit are enough to model the growth trend of software projects. However, the simulated set of files does not represent any features of the actual software system except size. An extension of the model will show in the next sections how other aspects of the software system can be simulated. This allows extended research questions, which can be answered by the model.

4.2.2. Model to Simulate the Lifetime of Bugs

To model the lifetime of bugs we extend the previous model as follows. First, we add a new class representing occurring bugs in the software system under simulation. For modeling purposes, bugs, just like the files, can be considered as passive agents. Then, we introduce networks to model dependencies between agents. The entire model is depicted in Figure 4.6 and our publications [4, 8, 135] are based on this model.

A `Bug` is generated by the `SimulationContext`. For this, we assume that bug occurrences are uniformly distributed. After the instantiation of the `Bug`, it will be assigned to a almost randomly selected `File`. According to [95], a property that makes a possible assignment of a bug to a file more likely is the coupling degree of the file. We model this by computing `SimulationContext.errorProne()` based on the coupling degree, i.e., the number of links to other files, of the file in the change coupling network which will be also introduced for this model (see Section 4.2.2.1). This bug creation strategy models bugs as they are managed in ITSs and created by users, testers, or developers. We are using this approach due to missing links between bugs and bug-fix commits [65].

4.2.2.1. Dependency Networks

To model a software system more accurately, dependencies between involved agents have to be described. Repast Simphony provides networks for this purpose. Nodes

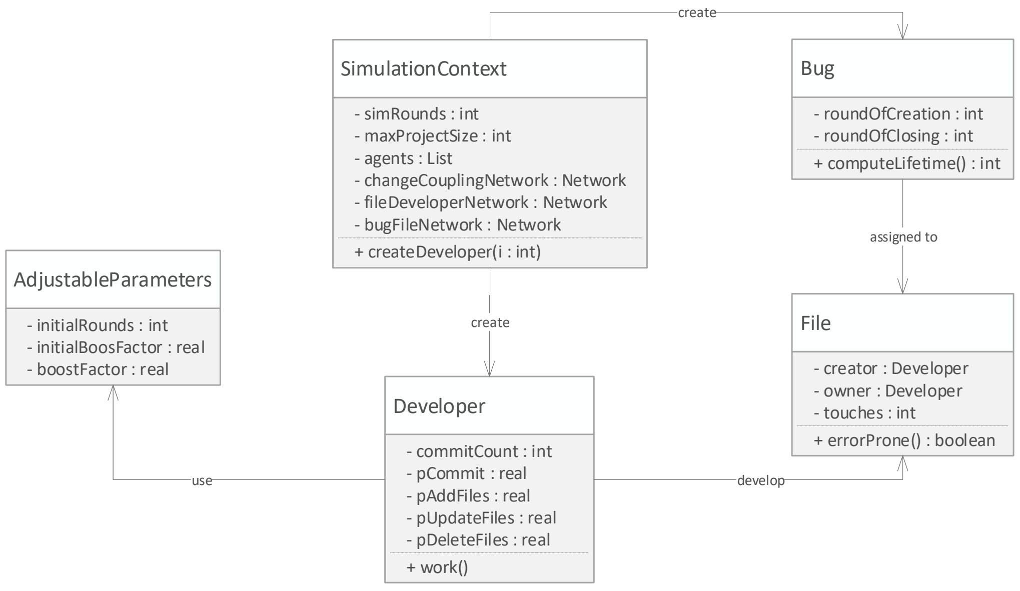


Figure 4.6.: Agent-based simulation model to analyze the lifetime of bugs.

represent the agents and these can be connected by weighted edges. Networks are managed by the `SimulationContext`. The following three networks are introduced.

File – Developer Network This network represents the dependency between a `File` and a `Developer`. The first edge connected to a file is created when a developer creates the file. Further edges are created when a developer modifies a file that has not previously connected to the developer. If a developer modifies a file and there already exists an edge between them, then the weight of this edge will be increased. The state of a file can change with every commit. An example of the file – developer network is depicted in Figure 4.7. Such a network provides the following properties

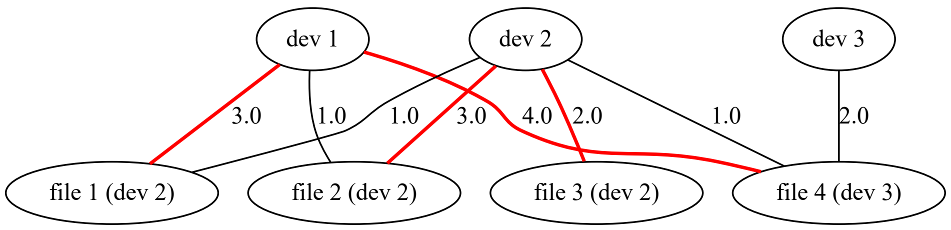


Figure 4.7.: Example for a file – developer network. Touches are represented as weight of an edge. The red marked edges represent the owner of a file. Behind the filename is named in brackets the creator of the file.

of a file.

Owner: the owner references the developer that changed the file at most and is independent of the creator. Therefore, the ownership of a file is dynamic and can be taken over by other developers over the time.

Number of authors: The number of different authors is represented by the degree of the file in this network.

Number of touches: The number of touches represents how often this file was part of a commit. It can be calculated by summing up the weights of all connected edges.

Bug – File Network After a `Bug` is created and the `File` to assign is selected, both agents will be linked by an edge in this network. The edge contains information whether a bug is closed or not. Thus, the bug does not have to be deleted and we can reopen it, if required. An example for a bug – file network is shown in Figure 4.8.

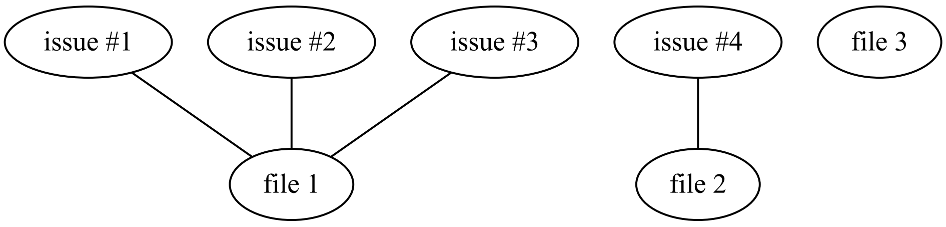


Figure 4.8.: Example for a bug – file network. Any created issue is assigned to an already existing file.

Change Coupling Network This network describes dependencies between files that are changed several times together in a commit. In Figure 4.9 an example for a change coupling network is presented. Such a network provides the change coupling degree of a file, which is used among other things for the determination of error prone files [95]. The change coupling network creates clusters of files that are changed often together. According to [2], files of one cluster are semantically related. Therefore, this network is the most precise representation of the software under simulation of this model.

The algorithm presented in Figure 4.10 creates the change coupling graph. The graph only contains edges between files that are changed together at least twice. To ensure this condition, we store an edge for the first change in the set E with the weight set to 1. The weight is the function $w : E \rightarrow \mathrm{N}^{+}$. In the next commit that changes both files the edge will be added to the coupling graph G and the weight gets increased. Afterwards, additional changes only increase the weight. Commits that

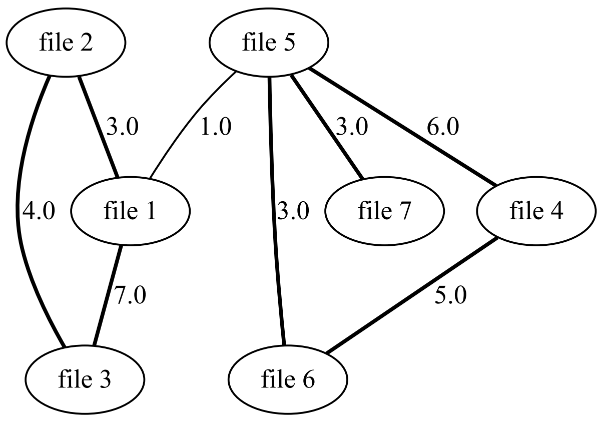


Figure 4.9.: Example for a change coupling network. The weights of the edges represent how many times files are changed together in one commit. Bold printed edges represent the membership in a cluster.

change a huge number of modules, for example, merge commits or large renaming commits, are not considered due to the exit condition in line 4.

The behavior of the developers plays a crucial role in the evolution of the change coupling network over time. Except for the following two differences, it is exactly as described in Section 4.2.1.1. First of all, besides the number of files to be deleted and added, we also need the number of files to be updated. For this, we assume that this follows a geometric distribution as described for deletions and additions in Section 4.2.1.1. If we know how many files have to be deleted, added or updated, the files to be updated and deleted must be selected. This is a difficult task, because we have no information about the intention of the developers. The selection process is based on the first selected file which is randomly selected. Further files will be selected based on information about the first one, like the owner, and the dependency networks. The files selected this way represent the files that the developer changes in a commit.

Using the networks described above, in combination with the developers contribution, it can be described how open bugs can be fixed. When a developer applies a commit and the set of files to be updated contains a bug, this bug can be fixed with a certain probability. This probability depends on the ownership and the coupling degree of the file. It is most likely that the owner of the file fixes the assigned bug if the file has a low coupling degree. Furthermore, older bugs are fixed with higher probability than newer ones. This behavior allows the analysis of the lifetime of bugs.

For this model, considering the software at the file level is quite sufficient, since the representative of the software is the change coupling network. For some other models, for example, for modeling refactorings using graph transformations, a more

```
X {List of updated files in one commit}
E {Set of edges between files that are updated together}
G = (V, P) {The change coupling graph with P ⊆ E}
if |X| > 20 then
    return
end if
for i = 0 to |X| do
    for j = i + 1 to |X| do
        x ← X[i]
        y ← X[j]
        if (x, y) ∈ P then
            w_{x,y} ← w_{x,y} + 1
        else
            if (x, y) ∈ E then
                V ← V ∪ {x, y}
                P ← P ∪ {(x, y)}
                w_{x,y} ← w_{x,y} + 1
            else
                E ← E ∪ {(x, y)}
                w_{x,y} ← 1
            end if
        end if
    end for
end for
```

Figure 4.10.: Pseudo code for generating the change coupling network as presented in [8].

detailed representation of the software is necessary. As a next step, this model will be expanded in the following section to be able to answer more research questions concerning software evolution with the simulation.

4.2.3. A Detailed Model to Investigate Several Aspects of Software Evolution

The goal of this extended model is to answer more research questions regarding software quality. For example, questions can concern the development team structure [136], the experience of team members [137], or design patterns [138]. Mainly specializations of the classes `Developer` and `Bug` have been introduced and our publication [5] is based on the model depicted in Figure 4.11.

To analyze different team constellations, we added different developer types which differ in the effort they spent to the project measured in the number of commits and their bug fix probabilities. In our model, the developers are divided into maintainer,

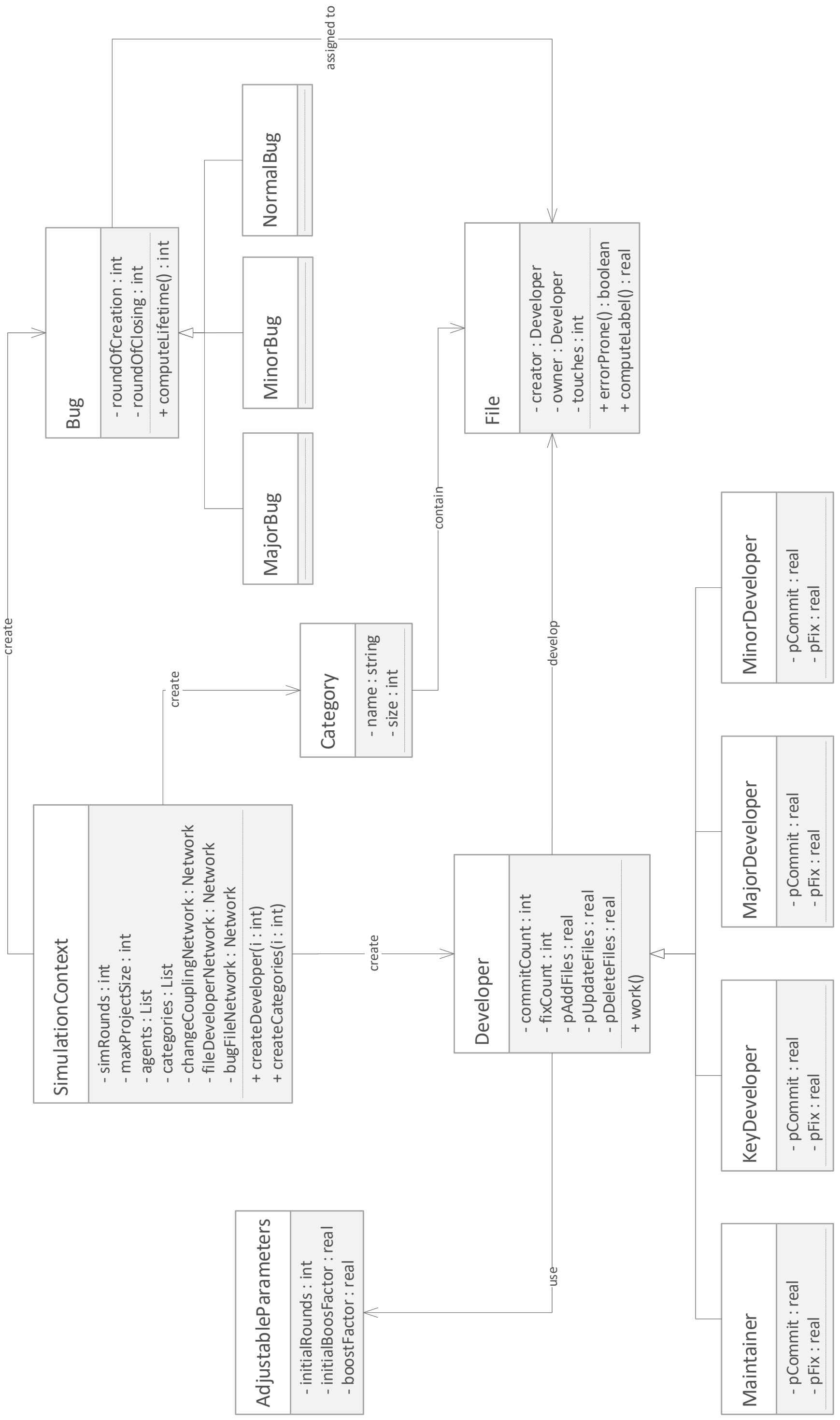


Figure 4.11.: Agent-based simulation model for different research questions.

key developer, major developer and minor developer. Maintainers are similar to key developers, but they perform a lot of maintenance work like bug fixes, in addition. Depending on the data from the mining (see Chapter 5), we can also simulate the usual developer types (e.g. [106]) core and peripheral. Thus, we can answer project specific research questions like: *Can we simulate effects like the loss of a key developer realistically?*

The process of creating and processing bugs has been improved by specializing the class `Bug`. The different types major, normal, and minor bug depend on their severity and in their number of occurrences. This, in combination with the new developer types, makes it possible to improve the bug fix strategy. For example, it is more likely that the key developer fixes a major bug than that the minor developer fixes a major bug.

Furthermore, we introduced a bug fix intention of developers. As mentioned in [116], corrective activities, such as bug fixes, generate tiny commits changing 1 to 5 files. Overall, about 80% of all commits are tiny ones. Therefore, instead of applying a commit as described above, a developer can have the intention to fix a bug. In that case, the set of files to be updated in this commit is selected as follows. First, one file with at least one bug is selected randomly. Afterwards, up to 5 files connected to the first one are selected also randomly. The selected bug can be fixed with a certain probability depending on factors like the ownership, the coupling degree of the file, and the developer type. This is the same decision process as when a normal commit contains a file with an assigned bug. With this model we can simulate the lifetime of the different bug types more precisely.

As additional class we introduced the `Category`. Categories represent folders or packages of the software in which files are grouped together in a logical context. At simulation start-up, the `SimulationContext` creates a given number of categories with a given size. Both parameters are determined by mining open source repositories. When a file is created, it will be assigned to a selected category with a certain probability. The probability of choosing a category depends on its size. This allows us to slightly adapt the file selection for a commit. The first file is selected randomly like before. For subsequent selections, we can assume that files in a commit are likely to belong to the same category. This update behavior structurally improves the simulated change coupling graph. This is noticeable by the fact that the clusters of the change coupling graph can be simulated more realistically.

Concerning the quality of each file we compute a label depending on assigned bugs as follows. For each bug type (major, normal, and minor) values are set indicating how much a certain bug type should decrease the label. Let us consider the following factors: 0.825 for major bugs, 0.9 for normal bugs, and 0.98 for minor bugs. Then, for each bug assigned to a file, the corresponding factors are multiplied. Thus, a file without a bug has the label value of 1 and every assigned bug decreases this value. This allows us, for example, to evaluate the overall quality of the software under simulation concerning the bug fix behavior of different team constellations.

Nevertheless, the update behavior of the developer remains almost unchanged, which means that the selection of the first file to be modified still depends on chance. As a result, the structure of the change coupling graph is strongly dependent on coincidence and not on the intentions of the developers. To address this issue, we present an approach to model further intentions of a developer in the next section. These intentions influence the selection of the files of a commit, including the first file selected.

4.3. Modeling Refactorings based on Graph Transformations

To improve the structure of the simulated software graph, we reduce the randomness of the file selection process for a commit. Therefore, we introduce the developers intention to refactor the code. For example, if a developer works on code with low maintainability (e.g. a method with above average size), it can apply a refactoring to improve the quality of the code. The developer's decision process is depicted in Figure 4.12. This behavior follows the prominent BDI [3] approach, where developers formulate goals based on their beliefs and build plans to reach them. Beliefs are the current state of the software under simulation, represented as software and graph metrics.

Refactorings are a common approach to describe well defined code structure changes like described in Section 2.3. To model software refactorings we use graph transformations (see Section 2.4). For each refactoring a transformation rule has to be defined.

In order to create a plan for each considered refactoring, we need detailed information about the software state before the refactoring is applied. The application of a refactoring causes state changes of software entities like the change of metrics of classes and methods as well as structural graph changes. This required data is collected through mining software repositories.

To apply these transformation rules, we need a more detailed graph for the description of the software under simulation. The used graph as presented in Definition 1 was introduced in [7]. It is transformed by the work of the developers according to formulated transformation rules. Due to the simplicity, it is easy to extend this graph when required. For modeling inheritance, for example, an additional edge label representing links between classes that belong to an inheritance hierarchy can be introduced.

Definition 1. *Let $\Sigma = \{C, M\}$ be a set of node labels and $\Delta = \{mm, mc\}$ be a set of edge labels. The node types represent software entities: classes (C) and methods (M). Edges represent relationships between nodes: method memberships (mm) and method calls (mc). A graph over Σ and Δ is a System $G = (V, E, l)$, where V*

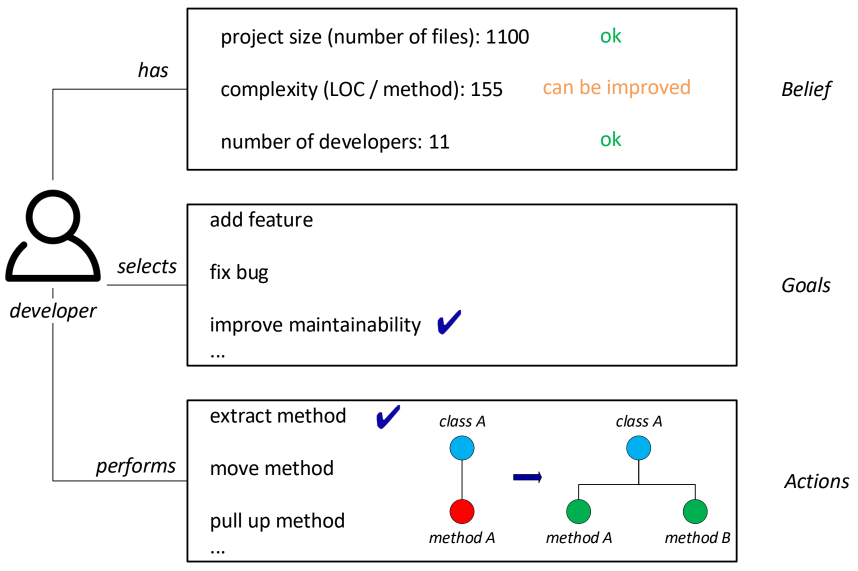


Figure 4.12.: Example for developer's intentions adapted from [7]. The developer works on a method that is hard to maintain because it has too many lines of code. To improve maintainability, the developer applies the refactoring *Extract Method*. Thus, parts of the origin method are moved to a newly created method that is called from the origin method.

is a set of nodes, $l : V \to \Sigma$ is a function that assigns a label to each node, and $E \subseteq V \times \Delta \times V$ is the set of edges.

To restrict the edge creation the *type graph* depicted in Figure 4.13 is used. The type graph is a generation specification such as the UML class diagram. For instance, member method edges link a class and a method as well as method calls can only occur between two methods.

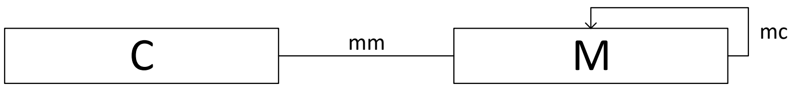


Figure 4.13.: Type Graph adapted from [7].

To integrate the new entities into the existing model, two networks need to be introduced. First, an undirected one that represents the membership of a method to a class. Second, a directed network to model method calls. To keep the model simple, we assume that there is exactly one class per file. An example for both networks is presented in Figure 4.14.

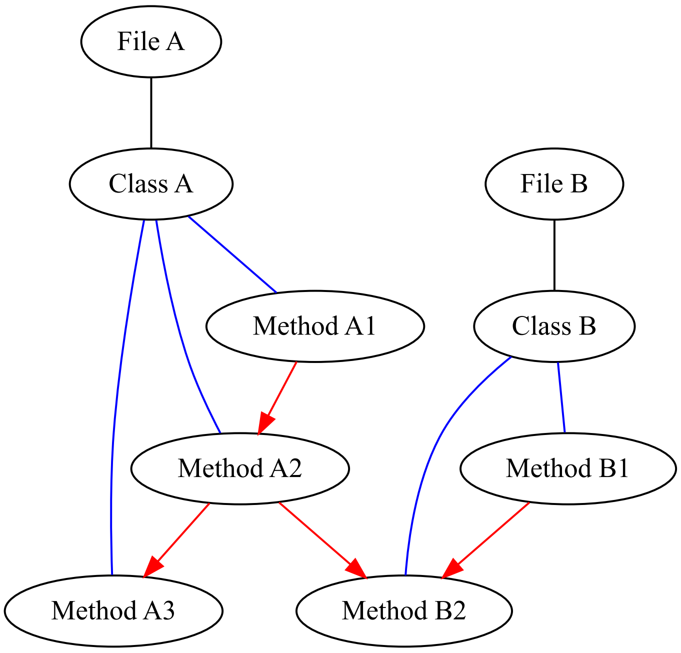
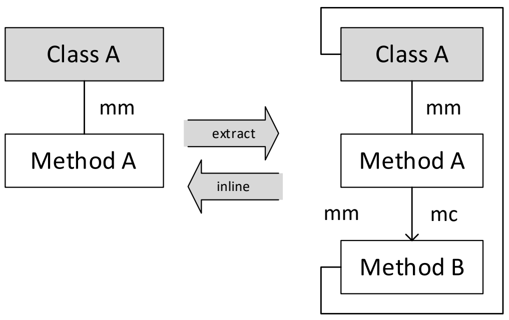


Figure 4.14.: Example for method call and method membership networks. The blue edges represent the method membership network. Method calls are represented by red edges.

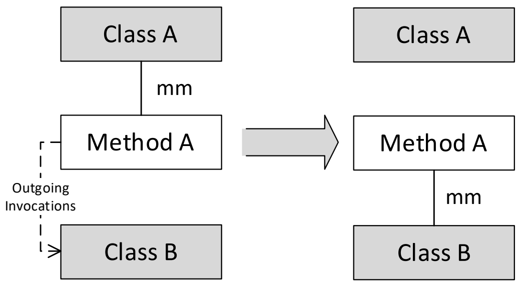
We introduced graph transformation rules for the three most frequently occurring refactoring types based on the mining process. The transformation rules are shown in Figure 4.15. The left hand side of the rule has to be replaced by the right hand side of the rule. In these rules the method calls, that are not important for the actual refactoring, are omitted. In the application of the rules it is ensured that all incident edges to a removed vertex are deleted as well. Therefore, no dangling edges occur after the transformation.

Since extract method is actually the opposite of inline method both refactorings are depicted in Figure 4.15a.

To find a match for the rule's left-hand side, not only the structure of the software graph is taken into account. Also appropriate software metrics are considered. These are, for example, the size measured in lines of code (LOC) for extract method (Figure 4.15a) and the coupling measured in number of outgoing invocations (NOI) for move method (Figure 4.15b). Furthermore, we assume that NOI of the origin class of a method will be reduced when a move method refactoring is applied.



(a) Extract/Inline Method.



(b) Move Method.

Figure 4.15.: Graph transformation rules for three software refactorings adapted from [7]. The left-hand side of the rule will be replaced by the right-hand side in the software graph. The dotted edge represents outgoing invocations, this means method calls to methods of the other class.

The modeling of refactorings reduces the randomness in the selection process of the files which are included in a commit. Furthermore, the scope of possible research questions is extended. How these models are implemented is described in the following.

4.4. Implementation Details and Execution of the Models

As mentioned before we are using the ABMS framework Repast Simphony (see Section 2.1.4) for modeling and simulation purposes in this thesis. For the implementation of the model Java is used and agents are implemented as POJOs.

The core class of all models is the `SimulationContext`. It serves as starting point of the simulation and after reading the current parameters, the context instantiates the configured number of each developer type and categories. The context knows all instantiated agents and the relationships between them. For the second property, it also creates and references the projections that represent the agents' relationships between each other. As projections, we only use networks in our models like the one used to model the change coupling graph. After the simulation model has been initialized, the simulation is executed for the configured number of rounds. Furthermore, the `SimulationContext` is responsible for the random generation of bugs during runtime. Each round an agent can execute its implemented behavior.

As active agents, we only have developers in all of our models. To mark the executable behavior, we use Java annotations. For example, the annotation `@ScheduledMethod(start = 1, interval = 1)` describes a method that is executed every round starting in the first round. If a developer executes its behavior like described in Section 4.2.1.1, the software evolves round by round. When new software entities are created by the developers, they are also added to the context.

To configure the simulation there are two different parameter sets available. Firstly, the core parameters to initialize the simulation model. They are generated for each analyzed project by an automated parameter estimation tool which is also developed as part of the work performed for this thesis. An example can be found in Section A.1. It contains the basic data for each project, for example, the maximum size of the project, the number of commits, the number of rounds to simulate, and the developers to instantiate. Furthermore, information about bugs, their fixes, and the categories of a project are available. Secondly, to adjust the behavior of the simulation based on the mined parameters, it is possible to change the parameters described in Section B.1 in the running application before each simulation run. With these parameters, for example, you can adjust the update behavior of the developers so that different growth trends can be simulated more realistically. The simulation at runtime is depicted in Section B.2.

Besides that, it is possible to start the simulation with any given commit or at any desired point in time. For this, the change coupling graph originating from the mining is read in (see Section A.2) and the simulation is initialized based on it. Thereafter, the simulation proceeds as described previously. For example, this feature can be used for validation purposes by comparing the real with the simulated change coupling graph at different times.

The simulated change coupling graph is exported as dot file[4] as well as the graph from the mining. The fact that real change coupling graph and simulated change coupling graph are in the same format makes it easy to compare both with each other.

Required Extensions for the Refactoring Model

For the implementation of the model of software refactorings, as described in Section 4.3, we added new passive agents for classes and methods to our existing model. Both have properties for size and complexity metrics. Furthermore, we added two projections for the modeled dependencies method membership and method call. Both are implemented as a network and the method calls use a directed network. These networks also provide coupling metrics such as the number of outgoing or incoming invocations.

Both networks together can be seen as a more abstract description of the software under simulation like an abstract form of the AST. Since we currently analyze only Java projects and there is predominantly one class per file, in the implementation a file corresponds exactly to a class.

[4]http://www.graphviz.org/doc/info/lang.html

In order to parameterize the model, parameters for general commit patterns, such as a bugfix or a feature add, as well as parameters for changes due to the application of refactorings are required.

The general commit patterns are needed to change the graph on every commit except for a refactoring. If a file in such a commit is added, a class is created and it is important to know how many methods have to be created and how the class is coupled to other classes. For updates of classes, the number of added, updated, and deleted methods as well as the changes to the coupling of the class are required. When a method is created or changed, the changes of the following metrics are considered: size measured in LOC, complexity measured in McCC (cyclomatic complexity by McCabe, see for example [38]), and the coupling measured in number of outgoing invocations (NOI) and the number of incoming invocations (NII). We use average values based on the data presented in Section A.3 for the parameterization. However, these average values are sufficient to simulate refactorings. We assume that all simulated metric changes of the abstract software graph are geometrically distributed like the number of files to be changed in a commit as described in Section 4.2.1.1.

For the parameterization of a refactoring, we need the changes of the considered metrics through the application of a refactoring. At this point, we also use average values of the metric changes shown in Section A.4. Furthermore, the metrics of the start class or method of the refactoring are considered in order to find an appropriate starting point of the rules left-hand side as described above. The number of found refactorings is used to decide how often a refactoring type is applied.

This resulted in the following changes to the contribution behavior of the developers. When a developer applies a commit, it is first decided what intention the developer has. If a developer has the intention to fix a bug or to add a feature, then the files to update will be selected as described in Section 4.2.1.1 and the abstract software graph will be changed as described above. However, if a developer intends to apply a refactoring, the files to be modified will be selected based on the transformation rule of the refactoring to be applied and the start metrics. To find a matching, the abstract software graph is searched. This limits the randomness of selecting the files to be modified in a commit. As a result, the more commits can be described by well-defined patterns, the less random the selection process of the files will be.

5. The Gathering of Parameters for Model Execution

Contents

In this chapter, we describe how we estimate the parameters required to instantiate our simulation models. First, we describe the overall process of mining tools based on SmartSHARK [79] in Section 5.1. Then, we introduce the automated parameter estimation tool in Section 5.2. Afterwards, the process to find refactorings and how the parameters for the transformation rules are estimated is presented in Section 5.3.

5.1. Overall process

Most of the developed mining tools in this thesis are based on SmartSHARK. Thus, these tools can use the data collected by several SmartSHARK plug-ins to compute the required simulation parameters. The overall mining process is depicted in Figure 5.1. As shown in this figure, the mining tools process data retrieved from

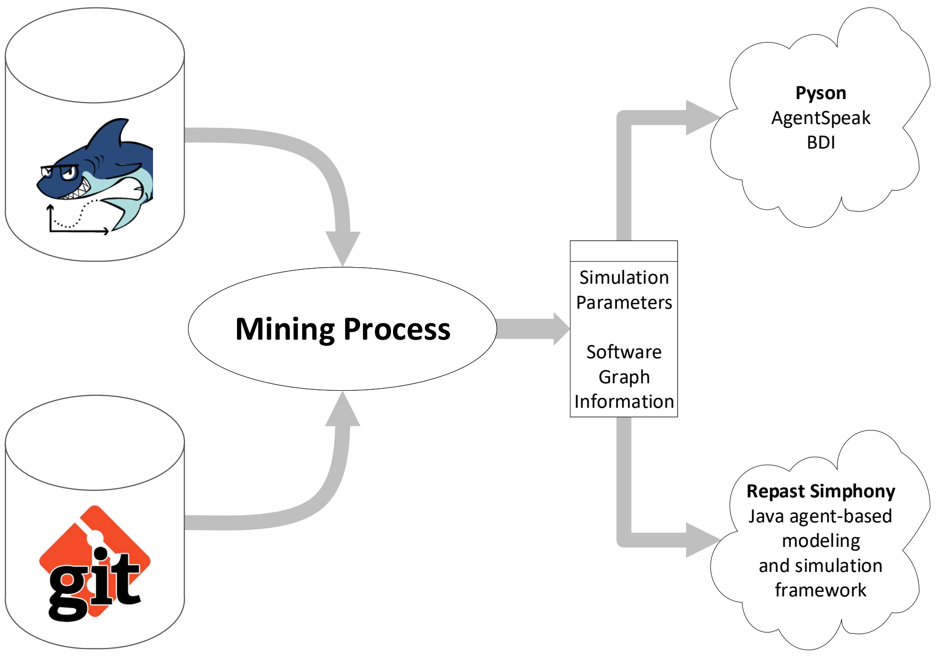


Figure 5.1.: Overall mining process using SmartSHARK.

the SmartSHARK database as well as in rare cases supplementary information from the git repository. The generated output are simulation parameters to instantiate the simulation models. These can be used by the implementation based on Repast Simphony presented in this thesis as well as by the AgentSpeak platform developed by our project partners [139, 140].

Since SmartSHARK is a collection of several tools, each of them for a specific task, that store their computed data in a MongoDB, we have to retrieve required data from this database as described in Section C.1. The following plug-ins of SmartSHARK are important for our mining work.

vcsSHARK: This tool is a VCS parser and provides information about commits, people, and files[1].

[1]https://smartshark.github.io/vcsSHARK

mecoSHARK: This plug-in collects static product metrics as well as code clone metrics[2].

issueSHARK: This tool gathers information from ITSs and thus provides data for the bug creation in the simulation[3].

These three plug-ins of SmartSHARK provide the necessary data for our mining tools to estimate simulation parameters. The data model created by the plug-ins is presented in Section C.2.

For the experiments in this thesis, which are based on SmartSHARK data, we are using a MongoDB hosted at the Institute of Computer Science at the University of Göttingen. This database contains data of several Apache Java projects analyzed with SmartSHARK.

5.2. Automated Parameter Estimation for Network Based Models

The mining framework presented in this Section gathers simulation parameters of projects that are analyzed with SmartSHARK before. It consists of the following four components as depicted in Figure 5.2.

Developer Information Provider: Since we are mainly interested in the developer's contribution to the project for simulation parameters, this tool collects all developers that are authors of at least one commit. Afterwards, two tasks are performed. First, as one and the same developer partially applies commits with different e-mail addresses, the identities of the developers are merged using an adapted identity merging algorithm. Second, developers, or more precisely the groups of developers belonging to one identity, are classified into different types.

Bug Information Provider: This tool investigates the *Issue* collection and provides information about the number of bugs that are created and fixed per year. Furthermore, all occurring priorities are mapped to the following three: *major*, *normal*, *minor*.

Commit Analyzer: To provide update probabilities for several commit types this tool investigates the *FileAction* information for a *Commit*. Thus, the number of updated, added, and deleted files per commit as well as the number of commits by type are available. This information is used to calculate the geometric distribution probabilities that describe the developer's contribution behavior. Currently, we use two commit types: an general commit type and a bugfix. The commit classification based on [141] is provided by SmartSHARK.

[2]https://smartshark.github.io/mecoSHARK/intro.html
[3]https://github.com/smartshark/issueSHARK

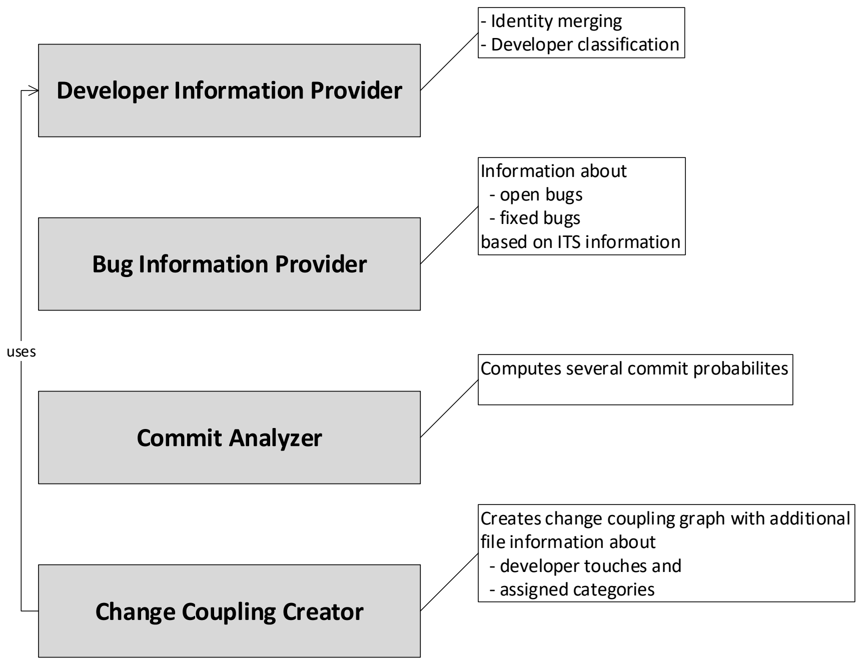


Figure 5.2.: Components of the overall process of automated parameter estimation.

Change Coupling Creator: This tool creates a change coupling graph for every year or any selected point in time as input for the simulation.

The computed data by the first three mining tools is exported as described in the example in Section A.1. An example for the exported change coupling graph can be found in Section A.2.

5.2.1. Developer Identity Merging

Our identity merge algorithm follows the algorithm presented in Figure 5.3 and introduced by Geominne et al. [69]. This algorithm describes a default merge algorithm which takes a set of people and returns a set of grouped identities. If people are merged depends mainly on the decision whether the people currently not added to an identity group can be merged with the person selected in the current run and, thus, be added to the identity group `tmpMerges` currently under construction.

The function `canMerge(m, i, t)` returns a boolean representing whether an identity `i` can be merged to the identity group `m` according to the parameter `t`. To decide whether an identity can be merged or not, we implemented an extended version of the simple algorithm presented in [69]. Since the person objects from the SmartSHARK database include the name, the email address and the username, we

```
1: $P$ {Set of persons $P = \{p_1, p_2, p_3, \cdots, i_n\}$}
2: $t$ {Similarity threshold, $t > 0$}
3: $identities \leftarrow$ emptySet() {The merged identities}
4: while isNotEmpty($P$) do
5:     $p \leftarrow$ getFirstElement($P$)
6:     $tmpMerges \leftarrow$ emptySet() {merged ID group for the current person p}
7:     insert($tmpMerges, p$)
8:     remove($P, p$)
9:     for all $x \in P$ do
10:        if canMerge($tmpMerge, x, t$) then
11:            insert($tmpMerges, x$)
12:            remove($P, x$)
13:        end if
14:    end for
15:    insert($identities, tmpMerges$)
16: end while
17: return identities
```

Figure 5.3.: General identity merge algorithm adapted from [69].

can use all these information to compare their identities. For each person we create labels using all available information. For the username we create one label. An additional label is created for the prefix of the e-mail address. Further labels are created based on the name and on the number of parts in which a name can be split. As separator a dot or a space is used. The following labels are created.

- Separators = 0: A label for the name is created.
- Separators = 1: Assuming that a name consists of tow parts, representing the first name f and the last name l, we create labels for the following combinations: $l.f$, $f.l$, lf, fl.
- Separators = 2: Assuming that a name consists of three parts, representing the first name f, the middle name m, and the last name l, we create labels for the following combinations: $l.f$, $f.l$, lf, fl, $l.m$, $m.l$, lm, ml, fml, $fm.l$.

To add a person to an identity group, all generated labels of the current person are compared with the labels of the current merge group `tmpMerges`. If two are equal in their first `t` characters, the person is added.

The comparison of the labels with each other is improved by the fact that a normalization method is used during the generation of the label. This means that all spaces are removed and uppercase letters are converted to lowercase letters.

To omit identities that are insignificant, we prevent the generation of labels for common email prefixes that probably do not represent a person like "mail", "dev_null", "dev-null", "noreply", or "github". Furthermore, the name "unknown" is omitted.

The authors of [69] figured out that a simple algorithm produces good results with the parameter `t` set to 3. This parameter selection also works well in our implementation tested against a manually merged set of people.

5.2.2. Developer Classification

The developers are classified into *core* and *peripheral* according to the prominent onion model as well as in *key*, *major*, *minor*, and *maintainer* like introduced for our simulation models.

The classification is based on the contribution of the developer, more precisely on the group of identities the developer belongs to. For this purpose, the contribution of each developer in a group is added up. This requires that the identities have been merged beforehand.

5.2.3. Change Coupling

To initialize the simulation at any time, we need a continuous change coupling graph. This means that changes are not forgotten after a certain time. Therefore, realistic developer information can be included in the graph. Before the change coupling graph is generated, there are two preparatory tasks to do.

First, the collection of analyzed commits is sorted ascending by the commit date. Second, all files in a commit that have been renamed are mapped to a representative. Thus, developer information can be assigned to a file. Afterwards, the change coupling graph is created as follows.

During the iteration over all commits, the change coupling graph is generated successively. For each commit, the following steps are executed:

1. For each file the representative is searched in the created map. This differs from the actual file only if it has been renamed. When we talk about files, always the representative is meant in the following.

2. It is checked whether the file is already contained in the current change coupling graph. If so, the developer information is updated. These are, for example, the number of touches as well as the owner. If this is not the case, a new file will be generated with initial information about the creator, its category, and a unique name.

3. Finally, the edges are created or updated. If an edge already exists between two different files of a commit, their weight is incremented by 1. Otherwise, a new edge of weight one is generated.

Only code files are considered for the graph generation. Such a created change coupling graph can be exported for any desired commit. By default, a graph is exported for each year. An example can be found in Section A.2.

5.3. Parameter Estimation for the Modeling of Refactorings

In this section, we describe how we collect parameters used for the refactoring model. First, we present parameter estimation for general commit patterns in Section 5.3.1. Afterwards, we present a framework which collects detailed information about the state of the software before a refactoring is applied and how the state changes when a certain refactoring was applied in Section 5.3.2. This framework does not use the SmartSHARK database as data source. Finally, we present a SmartSHARK plug-in named refSHARK, that replaces the previously described framework, in Section 5.3.3.

5.3.1. Parameters for the Description of General Commit Patterns

To find general commit patterns, each commit of a given project is analyzed as follows. First, the commit is classified into different activities. Second, the state of the software of the current commit is compared with the state of the parent commit.

For classification purposes, we use the four activities presented in [116]: *forward engineering* as a development activity as well as *re-engineering*, *corrective engineering*, and *management* as maintenance activities. The classification is key word based and for each activity a keyword list based on [116] is created. For example, commits that contain key words like "implement", "add", "request", and "new" in their commit messages represent forward engineering commits.

After the commit is classified, the code entity states (see Section C.2) of files, classes, and methods that are changed in this commit are used to read the desired metrics like lines of code (LOC) for the size of the analyzed software entity. To describe the changes, deltas of the desired metrics to the parent commit are calculated.

To parametrize the simulation model, we only consider the commit activities forward engineering (add feature) and corrective engineering (bugfix). The results can be found in Section A.3.

5.3.2. Framework to Estimate Parameters for Refactorings

To analyze the state of the software before a refactoring is applied as well as the changes made by a refactoring the framework depicted in Figure 5.4 is proposed.

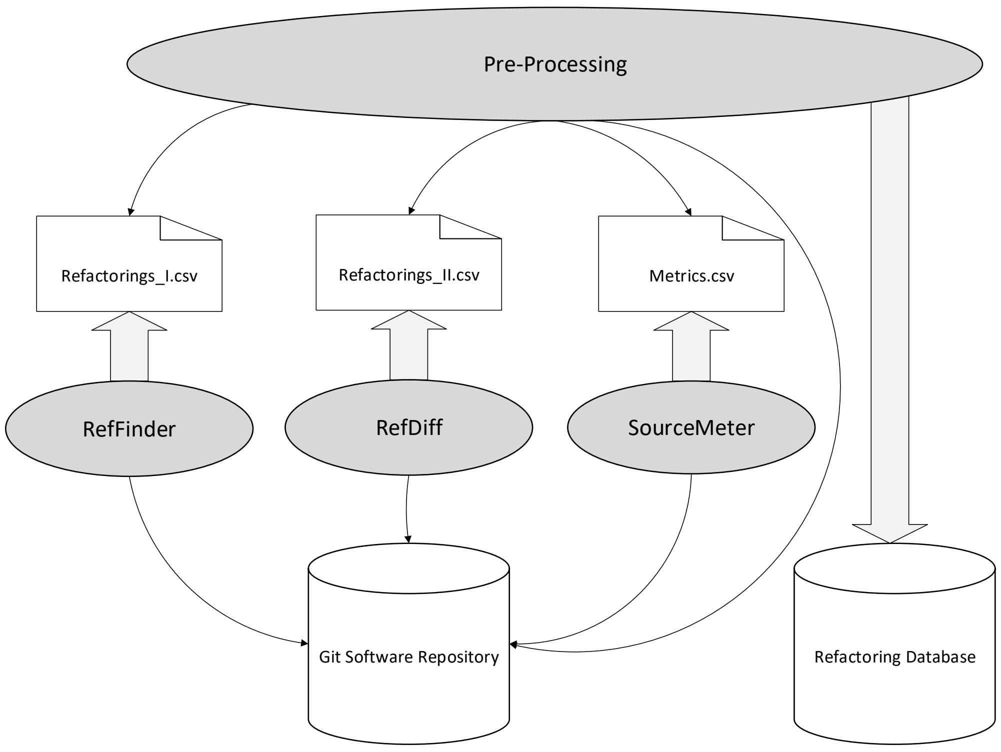


Figure 5.4.: A framework for finding refactorings in software repositories adapted from [7].

The framework is limited to git repositories as data source. It analyzes transitions between two consecutive code revisions by iterating over the commit history. In each transition the tool searches for refactorings. If at least one is found, the metrics of this transition as well as the corresponding deltas will be computed.

To find refactorings, the framework uses the tools Ref-Finder and RefDiff. Thus, we can compare the results generated by the two tools. Both are described in Section 2.3.

If a transition between two commits contains at least one refactoring, the desired start metrics and metric changes are analyzed. To retrieve the required source code metrics for classes and methods of both commits, the framework uses SourceMeter [77], a tool for static source code analysis.

The results of all tools are stored as text files. Unfortunately, naming conventions of the software entities are not unique. The framework provides preprocessing of this data to facilitate the analysis. The preprocessor searches the appropriate metrics in the output of SourceMeter for each refactoring found based on the output of Ref-Finder and RefDiff. Afterwards, metric changes are computed and the data set is complemented with information from the git repository. Finally, a data set for each found refactoring is stored in a MySQL[4] database.

Using this framework reveals efficiency and memory issues. It requires for mid-sized projects up to a week to analyze all commits on a regular dual core PC with 4 GB RAM. Furthermore, the comparison of the results of Ref-Finder and RefDiff showed that the results of RefDiff are more accurate for certain refactoring types. Based on the knowledge gathered during the implementation of the framework, we implemented refSHARK as described in the following section to improve the mining process of refactorings.

5.3.3. refSHARK to Estimate Parameters for Refactorings

To address the issues of the previously described framework, we have implemented a SmartSHARK plug-in named refSHARK. An advantage is that already calculated metrics by other SmartSHARK plug-ins stored in the MongoDB can be used. As a result, SourceMeter must not be executed at this point. Furthermore, to find refactorings in the revision history of git repositories we only use RefDiff since it has achieved more accurate results.

Since refSHARK requires commit and metric data, it is required that the plug-ins vcsSHARK and mecoSHARK are executed before. Afterwards, refSHARK iterates over the commits and searches for refactorings using RefDiff. If at least one refactoring is found, the tool searches for class and method metrics for the current commit and its parents. The corresponding code entity states, which contain the metric data, are stored in the MongoDB.

This plug-in stores the generated data in the MongoDB collections *Refactoring* and *RefactoringState*. The parent entries of `RefactoringState` refer to the superordinate elements in the AST. As an example, a class is parent of a method. However, the entries of `Refactoring.ParentStates` refer to the states of the parent commits in the git repository.

Such preprocessed data makes an analysis of found refactorings more efficient. Metric values before the refactoring was applied as well as delta values of the metrics can easily be retrieved by reading the corresponding code entity states.

The SmartSHARK plug-in refSHARK can also be found on Github[5].

[4]https://www.mysql.com/
[5]https://github.com/smartshark/refSHARK

6. Case Studies

Contents

This chapter presents the three case studies, that were conducted for this thesis. The first case study compares the different ABMs developed in this thesis. The second one investigates how project specific parameters for each analyzed project influence the simulation results. Furthermore, the simulation results of different simulation starting points are considered. The third case study concerns the mining and simulation process required for the simulation of refactorings.

6.1. Simulating Software Evolution using an Agent-Based Model

The experiments in this section show the variety of simulation models tailored towards different aspects of software evolution and their combinations. Several aspects of software evolution can be investigated with the proposed models. In general, the more complex the model, the more research questions concerning software evolution can be answered. With a specific question in mind, one can adapt a model to answer that question.

The overall process is similar for every considered model. First, the model is initialized with suitable parameters that are determined by the design of the model. Then, the model is executed and the results are compared with the expected ones from real software projects. If the results deviate too much, the model will be adjusted. Agent-based modeling follows a bottom-up approach. Therefore, small methods describing the behavior are adjusted to improve the overall state of the simulation. In rare cases, the parameters are also adjusted. Afterwards, the simulation is executed again. This iterative process is repeated until the simulated results fits the expected ones.

To validate the results, we use mainly easy to measure metrics like the number of commits performed by a developer type as well as graph metrics for network-based models. These metrics are compared with the ones from the real project over the time. For each model, different metrics describing the selected software evolution scenario are used for validation purposes.

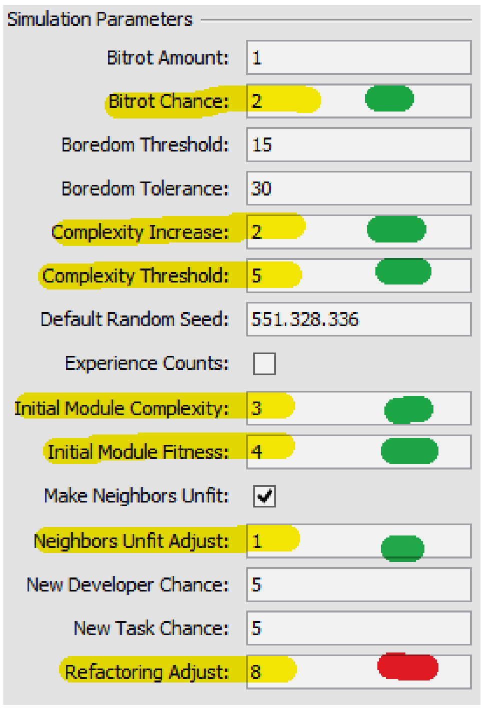
In the following, we have experiments with different setups and results for the different models. Finally, we present a general discussion of the different models.

6.1.1. A Grid-Based Model

The implementation of the grid-based model is based on [80] as described in Section 4.1. It helps us to experience how agent-based simulation works and what we can achieve with this kind of simulation in general.

Setup: The model is initialized with parameters according to the example given by the authors of [80]. This is the only model in which parameters are not based on mining open source repositories. This means, that quite general and no project-specific parameters are used here. The used parameter set is depicted in Figure 6.1. All of these parameters can be adjusted independently of each other for each simulation run.

For the evaluation of this model we are interested in the following characteristics by way of example: the project growth measured in the number of modules, the



Simulation Parameters

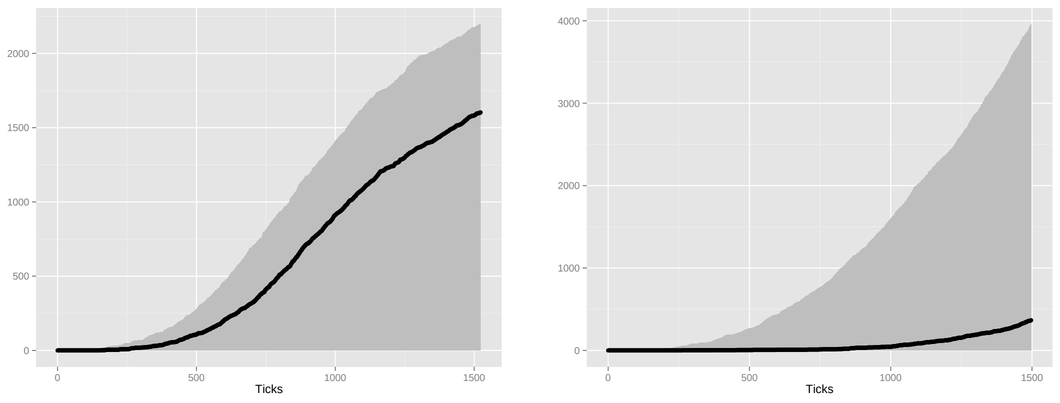
Bitrot Amount:	1	
Bitrot Chance:	2	🟩
Boredom Threshold:	15	
Boredom Tolerance:	30	
Complexity Increase:	2	🟩
Complexity Threshold:	5	🟩
Default Random Seed:	551.328.336	
Experience Counts:	☐	
Initial Module Complexity:	3	🟩
Initial Module Fitness:	4	🟩
Make Neighbors Unfit:	✓	
Neighbors Unfit Adjust:	1	🟩
New Developer Chance:	5	
New Task Chance:	5	
Refactoring Adjust:	8	🟥

Figure 6.1.: Parameter set for the grid-based model. The green marked parameters are decreased in comparison to the given specifications and the red marked ones are increased.

complexity of the entire software system and the fitness of the modules. To improve the results of this model, we do not change the behavior of the agents, but only change the parameters. For this, we decrease some parameters that increases the initial complexity or the complexity over time. Furthermore, we increase the refactoring parameter which ensures that the complexity of a module decreases after the application of a refactoring. These adjustments are also represented in Figure 6.1.

Results: The results, concerning the fitness of the modules as defined in Section 4.1, are depicted in Figure 4.3. With the given parameter set a lot of modules with low fitness occur. Using the adapted simulation parameters depicted in Figure 6.1, the entire system has a higher fitness. Similar observations can be made for the number of complex modules. With the default parameters there are many modules with high complexity as depicted in Figure 6.2a. The changed parameter set leads to a ratio as described in [80]. This means that about 10% of all modules are complex ones which is presented in Figure 6.2b.

Concerning the simulated growth trend, the growth simulated with the given parameters are more common as the one simulated with adjusted parameters. We also made this observation when trying to reproduce other properties of the software.



(a) Simulation results with a default parameter set.

(b) Simulation results with a adapted parameter set.

Figure 6.2.: Simulated complexity and growth of the grid based model. The number of complex modules is shown in relation to the total number of modules over time.

For various properties mostly different parameter sets are required. For this reason, all subsequent models are initialized with parameters based on the mining of open source projects.

Furthermore, the grid-based model does not allow the modeling of dependencies between the different entities. To address this issue and to model a more goal directed behavior of the agents, the following models are no longer based on a grid.

6.1.2. A Model without Dependencies

To analyze the growth of a software project no dependencies between the agents are required and the model described in Section 4.2.1 can be used. For example, the growth trend simulation in [4] is based on such a model.

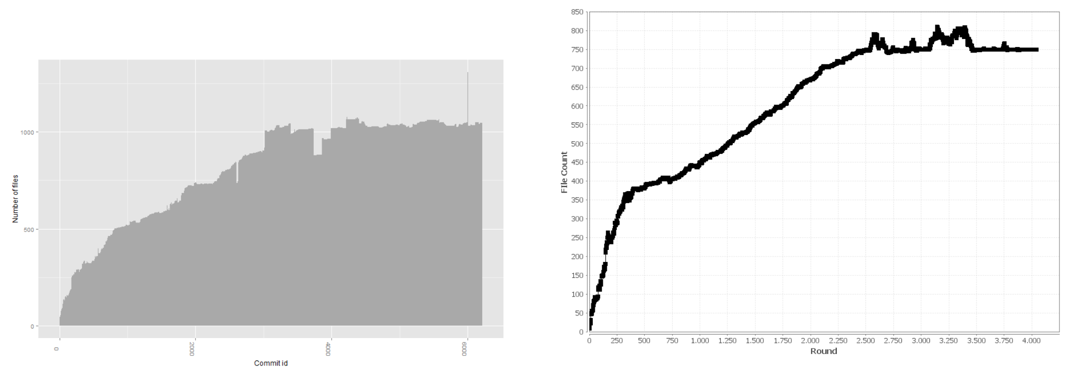
Setup: To initialize the model presented in Section 4.2.1, parameters to describe the contribution behavior of the developers (see Section 4.2.1.1) are required. Specifically, this means that we need to know how often a developer performs a commit and how many files are changed in a commit. Considering only the growth trend, changes mean added and deleted files. For this, the probabilities for the geometric distribution are calculated as described in Equation (4.1).

The mining process for this experiment is part of the work in [142]. As described in [4], all these parameters are gathered for the open source project K3b[1]. For

[1] https://kde.org/applications/multimedia/org.kde.k3b

mining purposes we used the framework presented by Makedonski [115]. With such an instantiated model only the developer's work is responsible for the evolution of the software under simulation.

Results: The results depicted in Figure 6.3 show that we can reproduce the growth trend of a real software project with the model presented in Section 4.2.1.



(a) Growth based on mining process. (b) Simulated growth.

Figure 6.3.: Simulated and mined growth trend of K3b adapted from [4].

This model is more or less limited to the simulation of the growth of a software project. To simulate other interesting factors of software evolution, we extend this model. A model extended with dependencies between the agents as well as more entities is evaluated in the following section.

6.1.3. A Network-Based Model

In this experiment, the network-based model described in Section 4.2.3 is evaluated. In doing so, the model for simulating the lifetime of bugs presented in Section 4.2.2 is also included in the evaluation since the former one is based on it.

This experiment is based on [5], where the following research question is analyzed: "Can we simulate effects like the loss of a core developer realistically?". To assess the state of the entire software under simulation, we consider the number of open and fixed bugs. Thus, the lifetime of bugs is an important factor for the evaluation of the entire software system under simulation.

Setup: To initialize the simulation model parameters for the contribution behavior and the bugfix frequency of the different developer roles are required. Furthermore, the bug introducing probability for each modeled bug type and the number of initial categories must be determined. The mining process for this experiment is based on the mining framework described in [115]. To adapt the simulation, only the parameters based on mining are changed for different runs.

For this experiment, three projects of similar size and duration are examined. However, these projects differ in the effort that the developers spent. A complete data set to instantiate the model comes from K3b[1] from previous studies. For the other two projects Log4j[2] and Kate[3], only the parameters for project size, number of developers of a certain type, the project duration and the number of initial clusters are determined. All other parameters are fixed and based on K3b. An overview of several parameters is presented in Table 6.1.

Project	Commits	Files	Developers	Duration
K3b	6142	1046	(1|1|6|116)	12
Log4j	8082	620	(1|1|6|13)	13
Kate	14282	681	(1|1|10|328)	11

Table 6.1.: Overview of project parameters.
For developers: $(core|maintainer|major|minor)$.
Duration: in years. Adapted from [5].

Results: From the mining perspective, essential parameters for simulating a core developer's loss are information about the team constellation as well as the bug introducing and fixing rates. The latter are used to monitor the impact on the software quality. According to [111], changes in the team constellation influence also the quality of the software.

The commit behavior of the developer types is shown in Figure 6.4 for K3b, in Figure 6.5 for Kate, and in Figure 6.6 for Log4j. The following heuristics are based on this results: core developers perform more than 20% of all commits; more than 25% of the commits of a maintainer are bugfixes; major developers apply more than 2% of all commits; minor developers perform less commits. The number of each developer type shown in Table 6.1 is based on this heuristics.

Since the change coupling graph represents the number of files that are changed together several times and files that are semantically related build clusters in this graph [2], we add the number of clusters to the parameter set of the simulation. We are more interested in clusters of higher dependency. Thus, we omit clusters with less then 5% of the nodes of the graph. The evolution of the number of clusters of the three projects is depicted in Figure 6.7. The number of larger clusters will be between three and six for the analyzed projects after a strong growth in the beginning.

[2]http://logging.apache.org/log4j
[3]https://www.kde.org/applications/utilities/kate

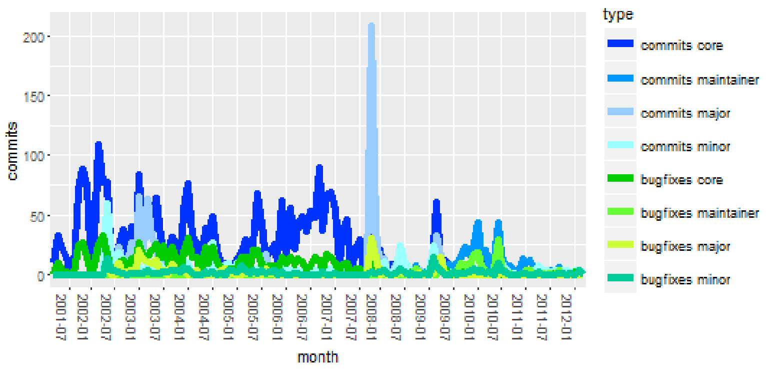


Figure 6.4.: Commits by developer type of K3b per month adapted from [5].

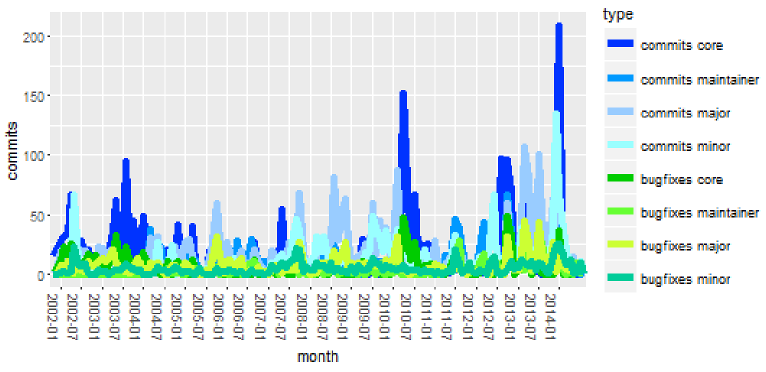


Figure 6.5.: Commits by developer type of Kate per month adapted from [5].

To model bugs, we use average values for overall reported and closed ones managed in ITSs. We consider the bug types *major*, *normal*, and *minor*. Other types occurring in the ITS are assigned to one of these three. For example, the reported and closed bugs of Kate are depicted in Figure 6.8. For the instantiation of the model, we use average values of all three projects. These are 0.87 reported and 0.81 fixed bugs per day which is synonymous with a round in the simulation.

With these parameters, we can instantiate the simulation models. First, we simulate the reference project K3b. Afterwards, we change particular parameters according to the mined data for each of the other projects and simulate them. Parameters to be changed are the project size and duration as well as the number of developers per type. To answer the question whether the loss of a core developer can be simulated, we perform two simulation runs for each project. First, a run without changes to the mined parameters. Second, a run where the core developer leaves the project after half of the duration. This results in 12% to 20% fewer bugs that are fixed due to the reduced effort.

The simulated growth measured in the number of files of this model fits well for the project K3b as depicted in Figure 6.9. The other analyzed projects have a similar

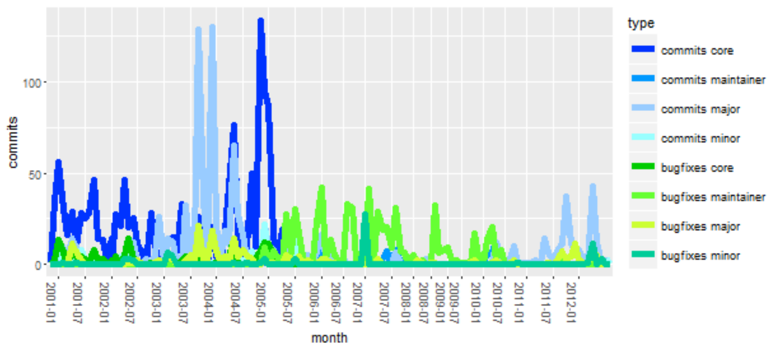


Figure 6.6.: Commits by developer type of Log4j per month.

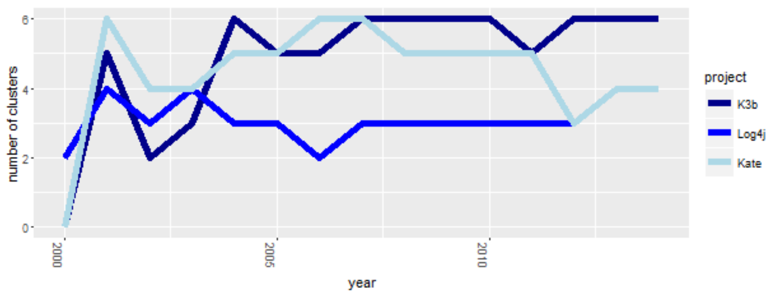


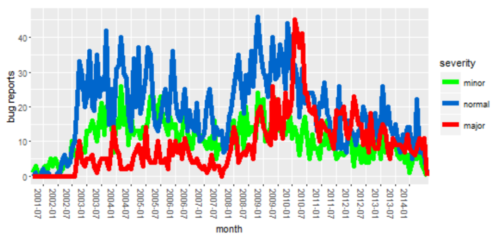
Figure 6.7.: Number of clusters over time.

growth trend that can be reproduced as well. The results of simulating projects with other growth trends can not be displayed with this setup. This is due to the fixed parameters based on the mining of K3b.

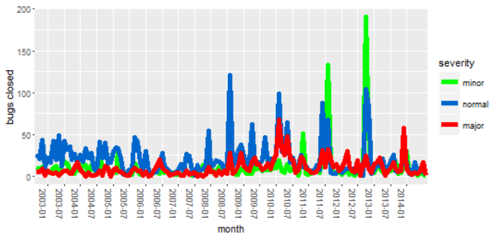
Furthermore, we analyzed in [8] how change coupling networks generated by an agent-based simulation evolves in comparison to the ones originating from the project history. This study is part of the work presented in [142]. The setup is similar to this experiment and K3b is used as the reference project. The simulation results are validated with results of Log4j. We compared several graph metrics like the coupling degree or the modularity. As a result, we figured out that the general evolution of file dependencies can be presented by a network-based simulation model. However, the modularity deviates in part from the reality.

6.1.4. Discussion

The experiments of this case study overall reveals that we can simulate aspects of software evolution using an ABM. The first thing we learned is that we do not need dependencies between the agents to simulate the growth of a project. Since such a simple model is not suitable for answering more complex questions, we have



(a) Reported bugs.



(b) Closed bugs.

Figure 6.8.: Reported and closed bugs of Kate.

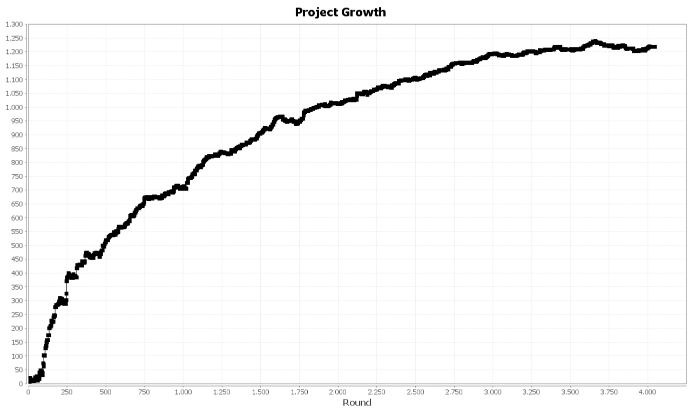


Figure 6.9.: Simulated growth of K3b.

introduced and evaluated network-based models. With these models we are capable to simulate the loss of a core developer or the lifetime of bugs as one example.

However, we also figured out that such an instantiated simulation model can only reproduce one growth trend similar to the growth of the project K3b as depicted in Figure 6.9. Furthermore, these experiments reveal that the random selection of files for a commit and the fixed parameters based on K3b influences the generated change coupling graph negatively. These two topics are covered in the following case studies.

6.2. Project Specific Parameters

In the experiments of the previous case study and in our previous work [4, 5, 8], we combined a fixed parameter set based on K3b with parameters for the respective project to instantiate the simulation. With this setup, we can only reproduce the of growth of projects similar to K3b. To address this issue, we propose two approaches. First, we gather a complete set of parameters automatically for each project and initialize the simulation model with it in Section 6.2.1. Second, we analyze how the results change when the model is instantiated with a real snapshot of the project at any point in time in Section 6.2.2.

6.2.1. Model Initialization with Project Specific Parameters

To instantiate the simulation model described in Section 4.2.3 with a project specific set of parameters, we use the automated parameter estimation tool presented in Section 5.2. The parameters for each project are structured like the ones shown in Section A.1.

Setup: For this experiment, we analyze for each growth trend depicted in Figure 4.5, two projects that are processed with SmartSHARK and their data is available in the MongoDB which is hosted at the institute. For each project the simulation parameters are computed by the estimation tool and the simulation is executed using the corresponding parameters. This setup should show that we can reproduce more occurring growth trends of real software projects as with the setups before.

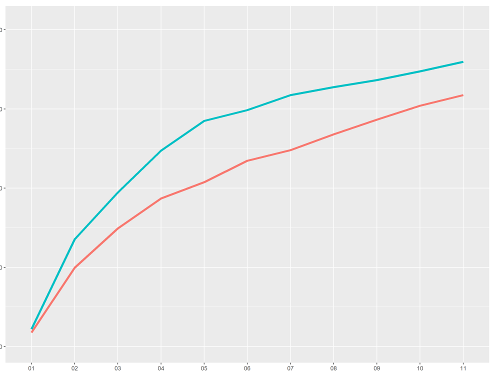
As projects we selected for a sub-linear growth trend at the end like depicted in Figure 4.5a *Zookeeper*[4] and *Directory Fortress Core*[5], for an approximately linear growth as shown in Figure 4.5b *Commons IO*[6] and *OI Safe*[7] as well as *Gora*[8] and *Nutch*[9] for a super linear growth at the end like depicted in Figure 4.5c.

For all these projects, we compare the simulated with the real growth over time. Afterwards, for selected projects the graph metrics presented in Section 2.4.2 are computed for the simulated and the real change coupling graph. By comparing the metrics over time, we can make statements about the quality of the simulated change coupling graph.

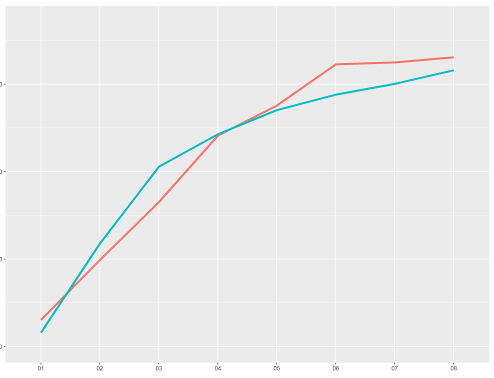
The projects for the detailed analysis are Gora, Zookeeper, and Directory Fortress Core. These three projects are selected because the parameters required for the simulation of refactorings are based on exactly these projects. Thus, we can compare the results of the following experiments with these results.

In order to improve the simulation results slightly, we adjusted the configurable parameters for each project as described in Section B.1 if required.

[4]https://github.com/apache/zookeeper
[5]https://github.com/apache/directory-fortress-core
[6]https://github.com/apache/commons-io
[7]https://github.com/openintents/safe
[8]https://github.com/apache/gora
[9]https://github.com/apache/nutch



(a) Growth of Zookeeper.



(b) Growth of Directory Fortress Core.

Figure 6.10.: Growth of projects over the time with a sub-linear trend at the end. The empirical progress is colored red and the simulated progress is colored blue.

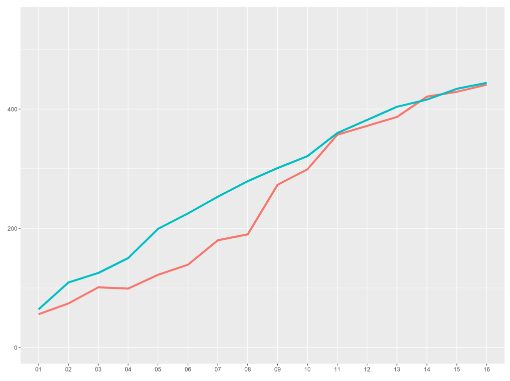
Results: The results depicted in Figure 6.10, Figure 6.11, and Figure 6.12 show that we can reproduce the different growth trends of real world projects with this simulation setup.

The evolution of the considered graph metrics is presented for Zookeeper in Figure 6.13, for Gora in Figure 6.14, and for Directory Fortress Core in Figure 6.15. It can be seen that the simulated modularity and the simulated degree deviate from the empirical values. This is especially the case for Directory Fortress Core, whereas Zookeeper have the best simulated values overall. Furthermore, it can be seen that the metrics density and diameter achieve good simulative values for all evaluated projects.

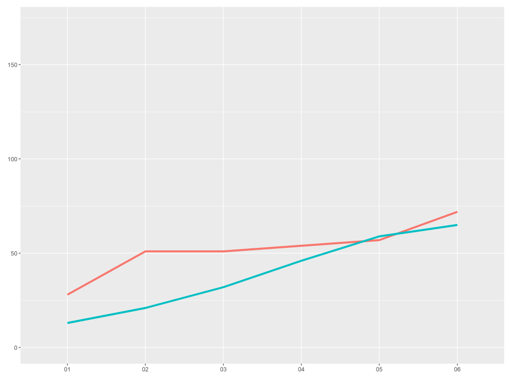
6.2.2. Model Initialization with Project Specific Parameters and Change Coupling Snapshot

This experiment investigates how the simulated results change when the simulation model is instantiated with a snapshot of the real change coupling graph of the analyzed project.

Setup: This setup is similar to the setup of the previous experiment presented in Section 6.2.1, but in this case the simulation starts at a selected year. To achieve this, the simulation model is not only instantiated with the project specific parameters but also with the change coupling graph of the desired year to start with. Then, the simulation is executed as usual. The change coupling graph is also generated by the automated parameter estimation tool described in Section 5.2 and can be generated for any point in time. An example of the change coupling graph can be found in Section A.2.

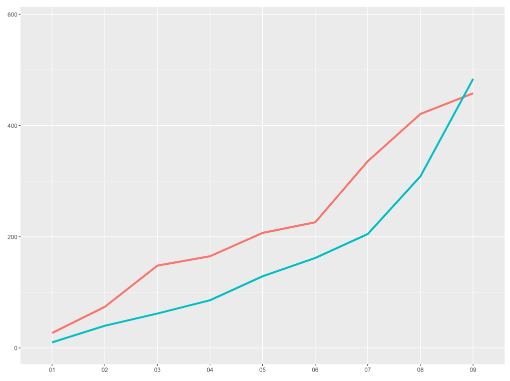


(a) Growth of Commons IO.

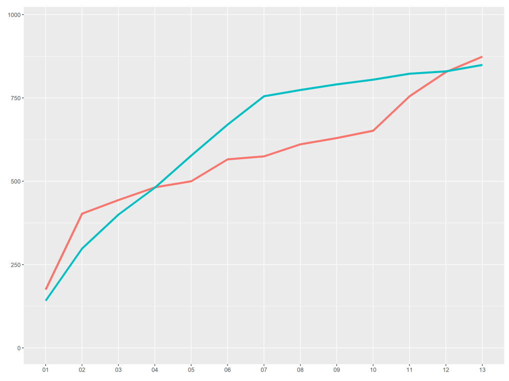


(b) Growth of OI Safe.

Figure 6.11.: Growth of projects over the time with approximately linear growth trend. The empirical progress is colored red and the simulated progress is colored blue.

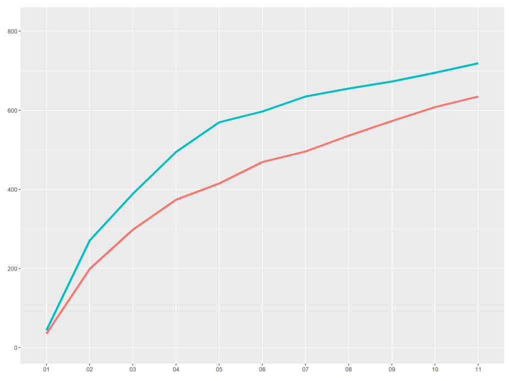


(a) Growth of Gora.

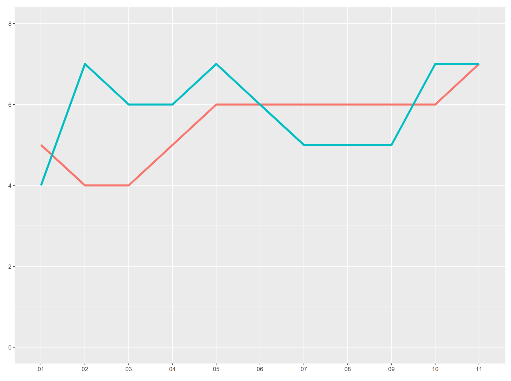


(b) Growth of Nutch.

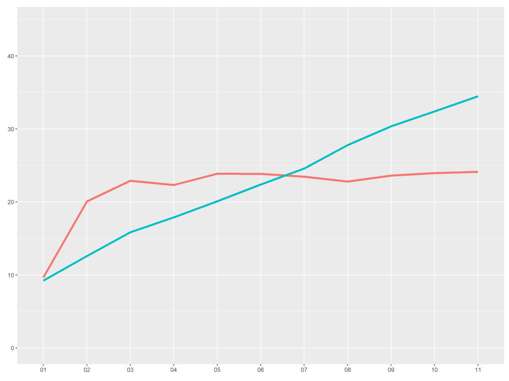
Figure 6.12.: Growth of projects over the time with a super-linear trend at the end. The empirical progress is colored red and the simulated progress is colored blue.



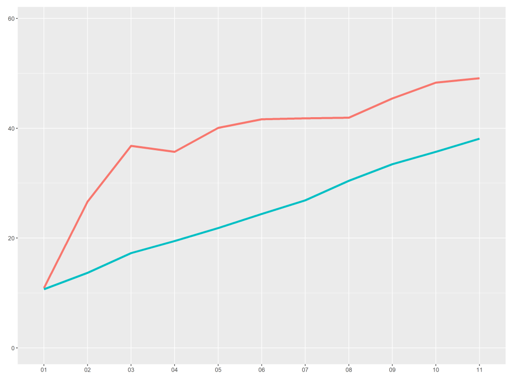
(a) Number of nodes (growth).



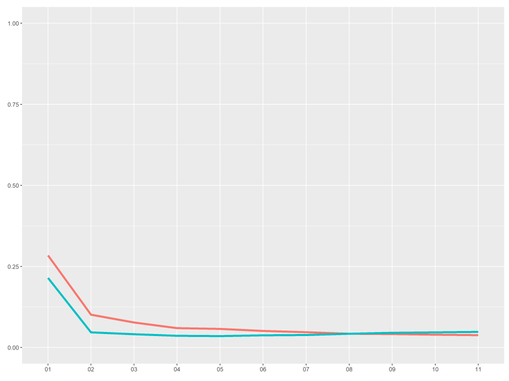
(b) Diameter.



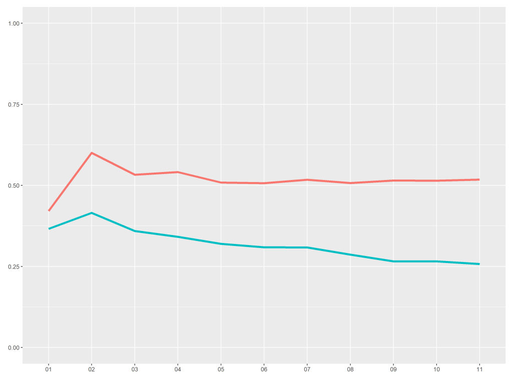
(c) Average degree.



(d) Average weighted degree.

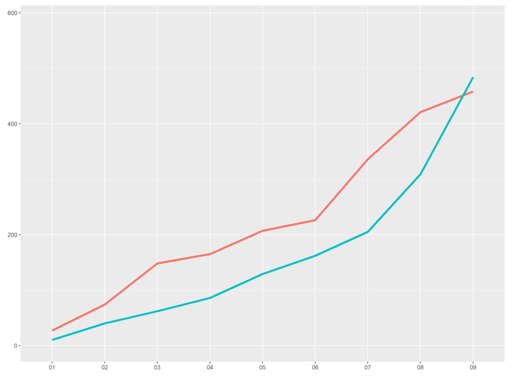


(e) Density.

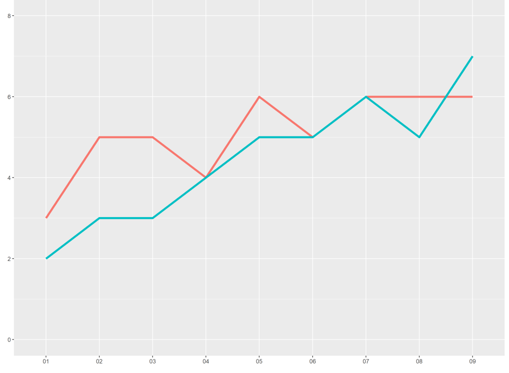


(f) Modularity.

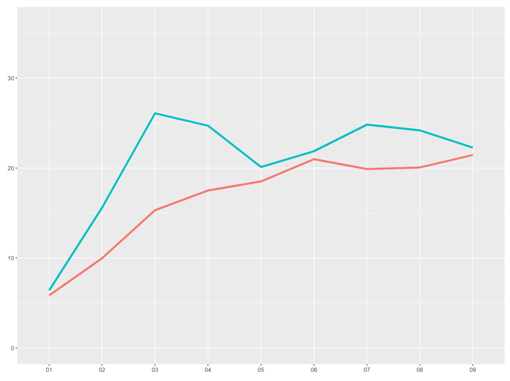
Figure 6.13.: Graph metrics of Zookeeper over the time. The empirical progress is colored red and the simulated progress is colored blue.



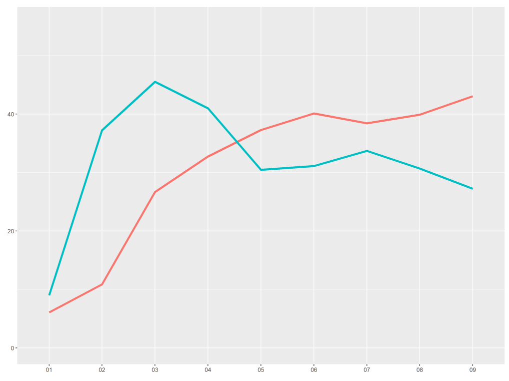
(a) Number of nodes (growth).



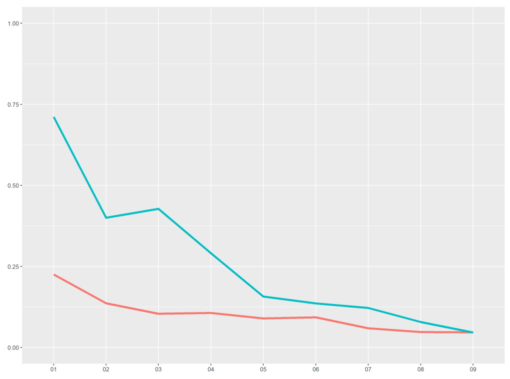
(b) Diameter.



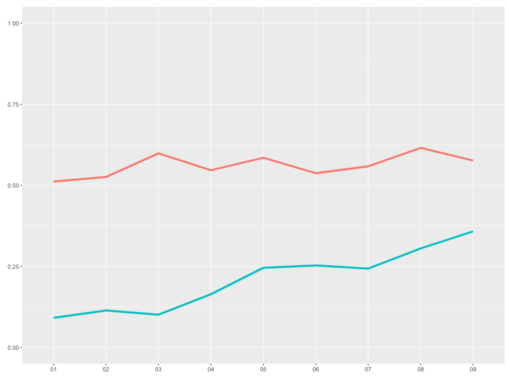
(c) Average degree.



(d) Average weighted degree.

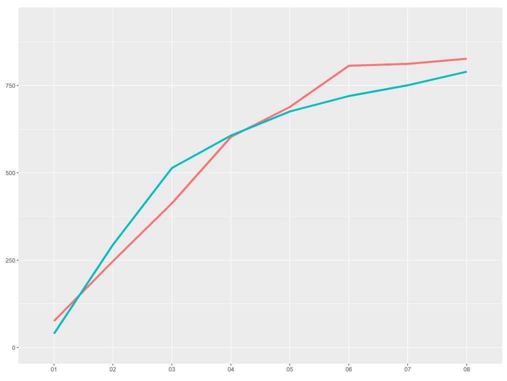


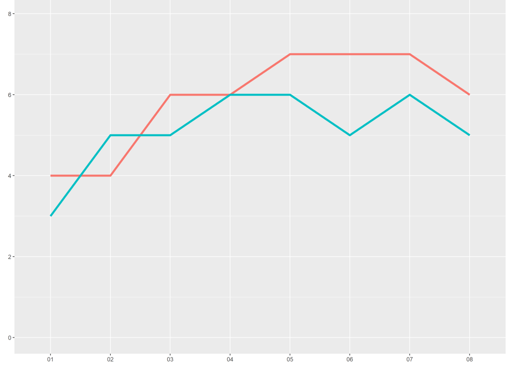
(e) Density.

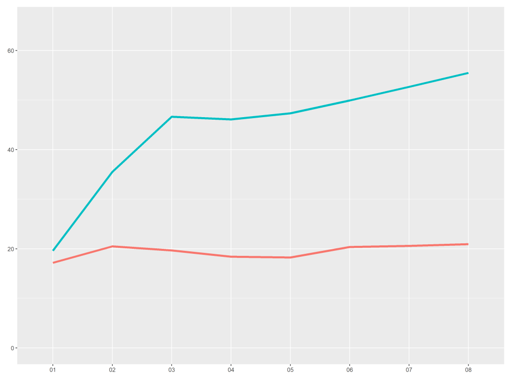


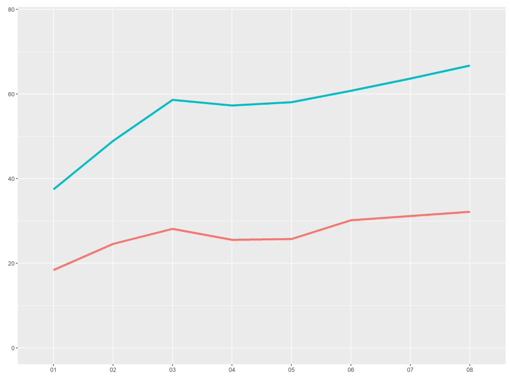
(f) Modularity.

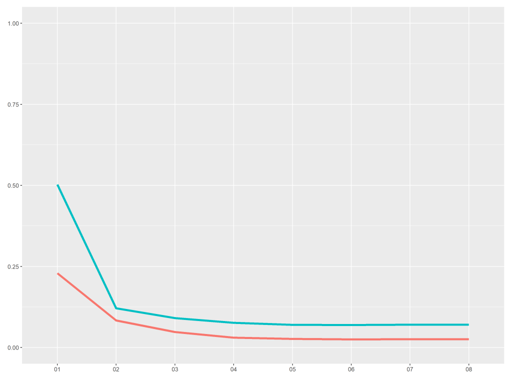
Figure 6.14.: Graph metrics of Gora over the time. The empirical progress is colored red and the simulated progress is colored blue.


(a) Number of nodes (growth).


(b) Diameter.


(c) Average degree.


(d) Average weighted degree.


(e) Density.

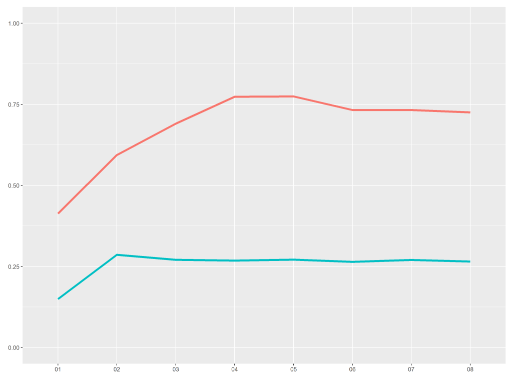

(f) Modularity.

Figure 6.15.: Graph metrics of Directory Fortress Core over the time. The empirical progress is colored red and the simulated progress is colored blue.

For this experiment, the simulation model for all analyzed projects is instantiated after about one third of the project duration. The projects to analyze are Gora, Zookeeper, and Directory Fortress Core. These three projects are selected to compare the results with the results of other experiments based on these projects.

Results: The evolution of the analyzed graph metrics is presented for Zookeeper in Figure 6.16, for Gora in Figure 6.17, and for Directory Fortress Core in Figure 6.18. In th figures one can see, that the simulation based on an initial snapshot improves most graph metrics. In particular, the simulated degree has significantly improved. An exception, however, is the growth of Directory Fortress Core as depicted in Figure 6.18a.

6.2.3. Discussion

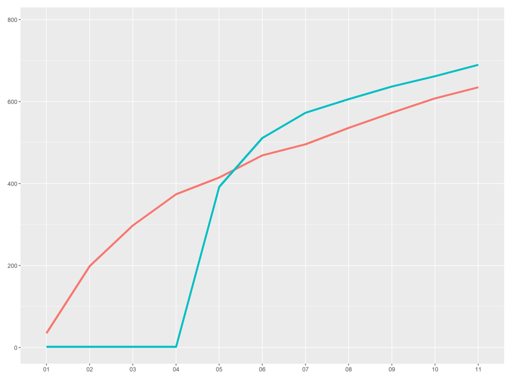
We figured out two important insights in the presented experiments of this case study. First, with a project-specific parametrization of the simulation model, we can simulative reproduce the usual growth trends. Second, a simulation model initialized with a snapshot of the change coupling graph can simulative reproduce the progression of the graph. For this, the simulation is started approximately after one third of the project runtime and the graph metrics are used for statements about the quality of the simulated graph.

6.3. Mining and Simulating Software Refactorings

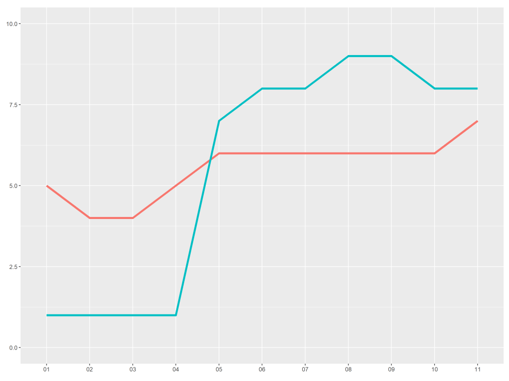
In this case study, we first analyze the feasibility of modelling refactorings using graph transformations in Section 6.3.1. Afterwards, we evaluate the integration of refactorings into our simulation model for software evolution in Section 6.3.2.

6.3.1. Feasibility of Refactoring Simulation

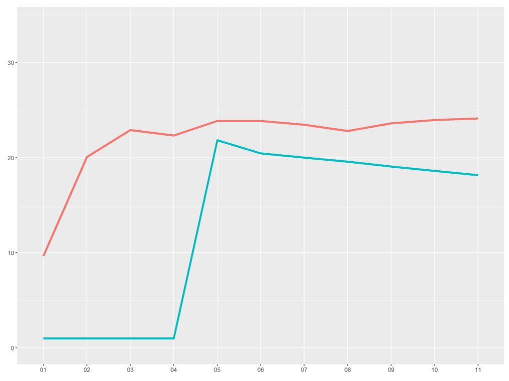
In order to make a statement about whether refactorings can be simulated using agent-based simulation and modeled by means of graph transformations, we must be able to find suitable parameters. To find these, we use the mining framework described in Section 5.3.2 for this experiment. After determining the parameters, a transformation rule can be created for each refactoring to be simulated (see Section 4.3).



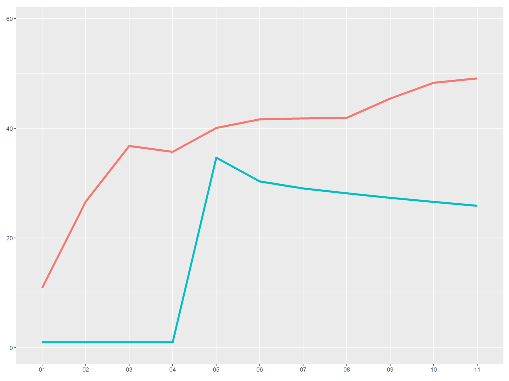
(a) Number of nodes (growth).



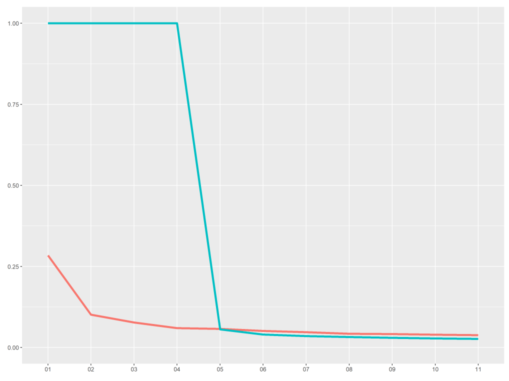
(b) Diameter.



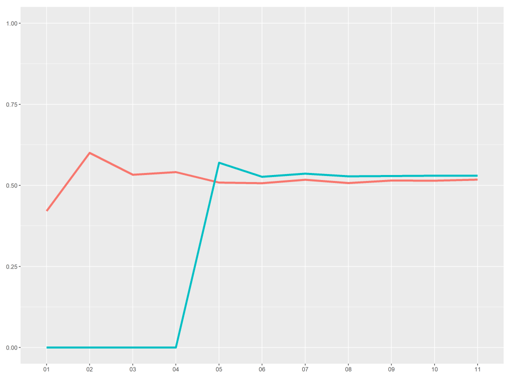
(c) Average degree.



(d) Average weighted degree.

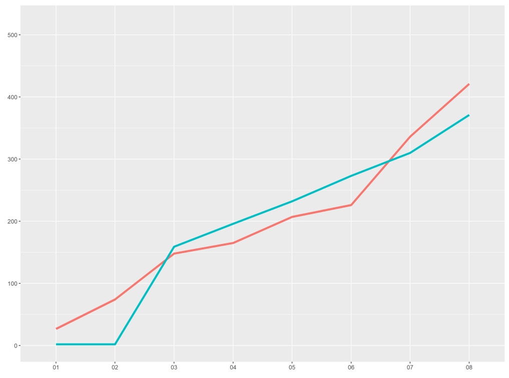


(e) Density.

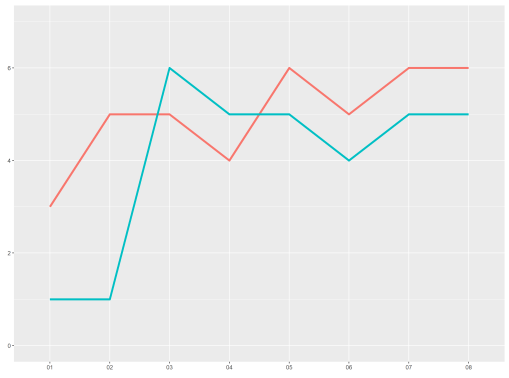


(f) Modularity.

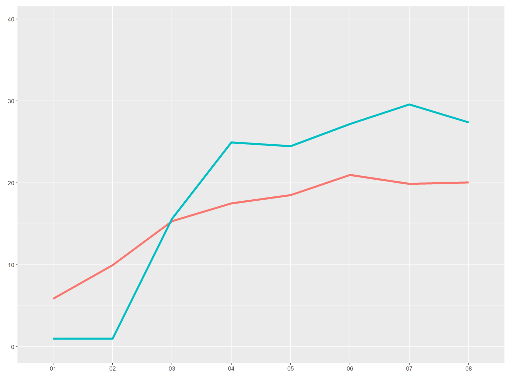
Figure 6.16.: Graph metrics of Zookeeper for snapshot initialization. The empirical progress is colored red and the simulated progress is colored blue.



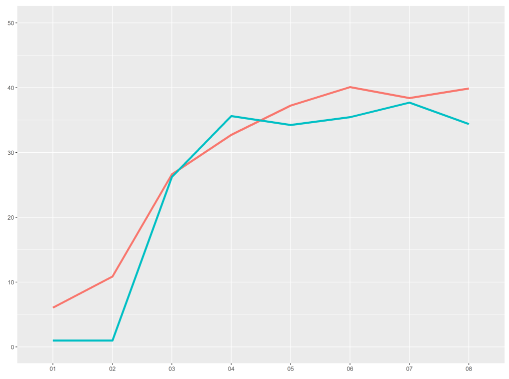
(a) Number of nodes (growth).



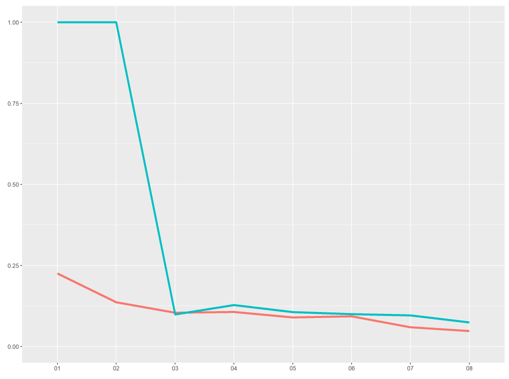
(b) Diameter.



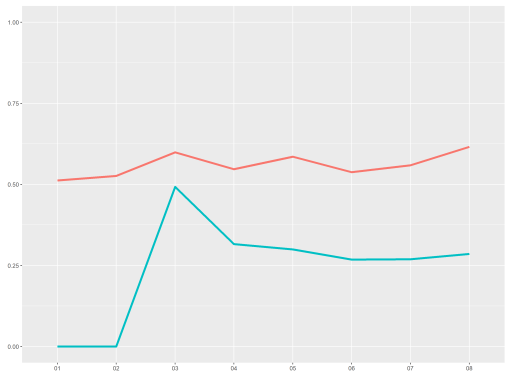
(c) Average degree.



(d) Average weighted degree.

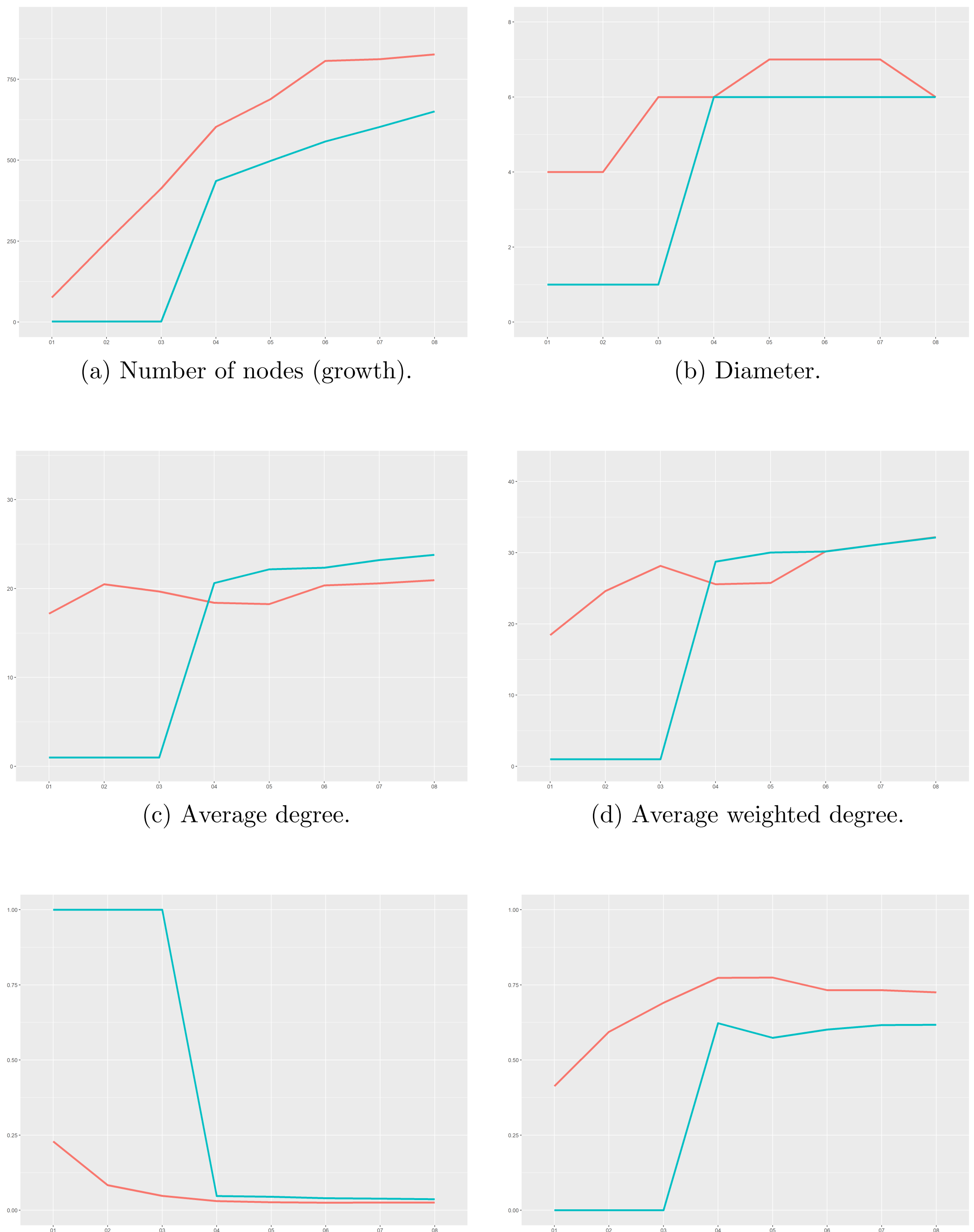


(e) Density.



(f) Modularity.

Figure 6.17.: Graph metrics of Gora for snapshot initialization. The empirical progress is colored red and the simulated progress is colored blue.



(a) Number of nodes (growth).

(b) Diameter.

(c) Average degree.

(d) Average weighted degree.

(e) Density.

(f) Modularity.

Figure 6.18.: Graph metrics of Directory Fortress Core for snapshot initialization. The empirical progress is colored red and the simulated progress is colored blue.

Setup: This study is based on [7]. Parameters to instantiate the refactoring model are gathered using the mining framework described in in Section 5.3.2. To find refactorings between two revisions of open source projects, the framework uses the tools Ref-Finder and RefDiff in parallel. Thus, we can compare the results of both tools. As projects we analyze *JUnit*[10] – a unit testing framework for Java, *MapDB*[11] – an embedded database engine for Java, and the *GameController*[12] – used, e.g., for several RoboCup competitions. Based on the parameters computed for these projects the refactoring model is instantiated and simulated. Furthermore, the model is initialized with a snapshot of the code base representing one revision in the source code repository. This means for this experiment that only refactorings are executed as commits and we only have one developer that performs refactoring after refactoring. For simulation purposes the simulation platform[139] developed by our project partners is used.

Results: The parameters to parameterize the simulation model are depicted in Figure A.1. The results show that Ref-Finder finds significantly more refactorings of the type move method than RefDiff do. This finding corresponds to the results of the authors of [45]. They figured out that Ref-Finder has a high number of false positives for some analyzed projects, in particular for the refactoring type move method. For this reason, we only use RefDiff for further examinations.

Furthermore, the mining process reveals that only a small amount of methods are moved to already existing classes when the refactoring move method is applied. Thus, in many cases methods are moved to newly created classes. The results are presented in Table 6.2.

	Tool	
Project	Ref-Finder	RefDiff
JUnit	5,6 %	22,5 %
MapDB	34,9 %	28,0 %
GameController	9,6 %	0,0 %

Table 6.2.: The amount of applied refactorings of the type Move Method where the method is moved to an already existing class.

This result justifies an extension of the transformation rule for the refactoring move method from Section 4.3 as depicted in Figure 4.15b. The proposed new transformation rule for move method is shown in Figure 6.19.

[10]https://github.com/junit-team/junit4
[11]https://github.com/jankotek/mapdb
[12]https://github.com/bhuman/GameController

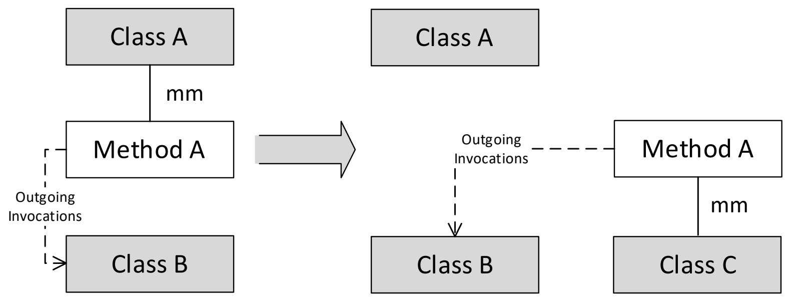


Figure 6.19.: Alternative transformation rule for Move Method adapted from [7]. The method A is moved to a new created class C.

This setup is restricted to the simulation of software refactorings. The transformation rules are implemented and they work as expected. This means, that the complexity decreases over time. For further investigations we need to integrate refactorings into the simulation model for software evolution as introduced in the following section.

6.3.2. Integration of Refactorings to a Simulation Model for Software Evolution

In this experiment, we analyze how the change coupling graph changes when refactorings are simulated. To integrate the refactoring model into the model for software evolution, we need to know how the abstract software graph required for the refactoring model changes when commits like a feature add or a bugfix are applied. This knowledge is required to let the software graph also evolve over time.

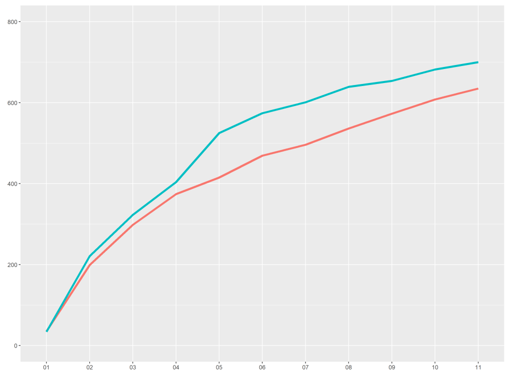
Setup: To get the required parameters for commit patterns like feature adds or bug fixes, we use SmartSHARK as described in Section 5.3.1. The mining process turned out to be time and memory intensive. Thus, we have only parameters for three projects as presented in Section A.3. The metric changes caused by a refactoring are based on the results of refSHARK (see Section 5.3.3) and can be found in Figure A.2. For each commit pattern we use fixed values based on the mined average.

To execute the simulation, the model is first initialized with the project specific parameters like in the previous experiments. The number of refactorings to apply is based on the mined data. Each time a refactoring is applied the transformation rule is applied and the simulated software metrics change as the mining data states for the rule. If a bugfix or a feature add is applied, the files for the commit are selected as without refactorings and the metrics of the simulated software graph are changed as described for the applied commit pattern.

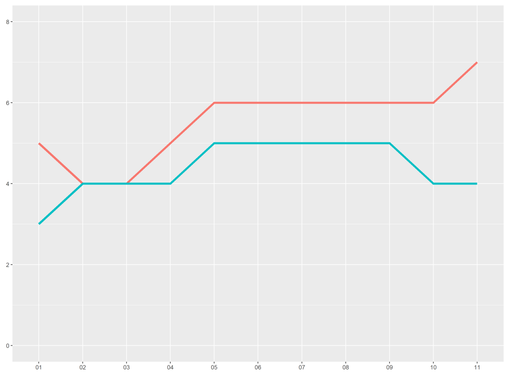
Results: The evolution of the analyzed graph metrics of the change coupling graph is presented for Zookeeper in Figure 6.20, for Gora in Figure 6.21, and for Directory Fortress Core in Figure 6.22. It can be seen that apart from growth, no further metrics improve.

6.3.3. Discussion

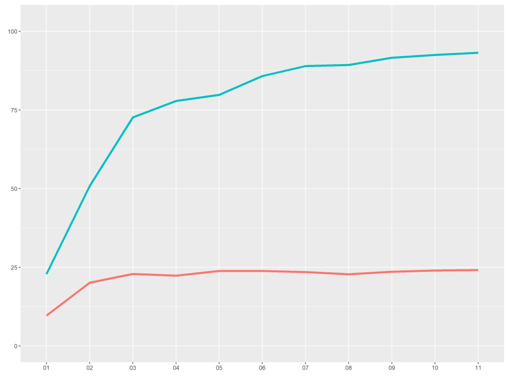
In this case study we show that we can figure out how the application of a refactoring change the software metrics. Furthermore, the approach to use graph transformation for the modeling of refactorings can be integrated in our agent-based simulation model for software evolution. However, to evaluate wether the simulated change coupling graph can be improved we require more commit patterns for further refactorings and other commits.



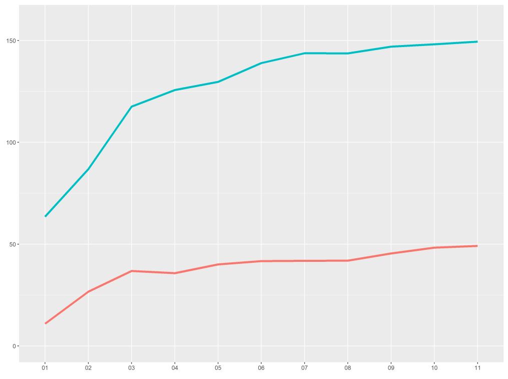
(a) Number of nodes (growth).



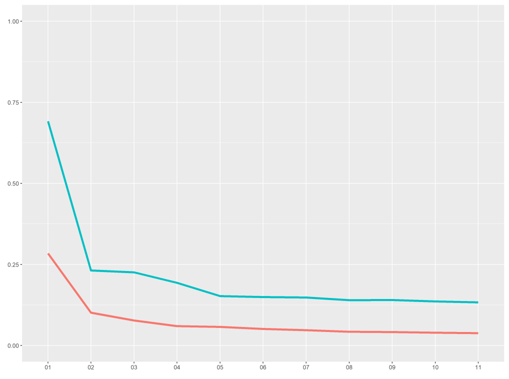
(b) Diameter.



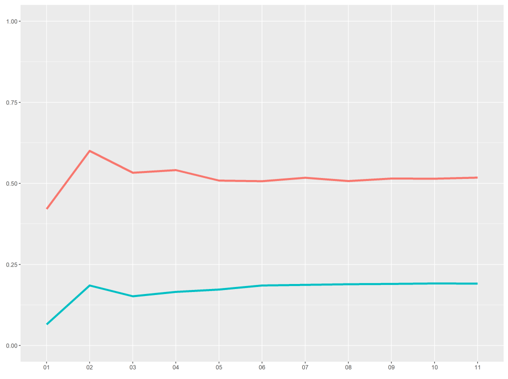
(c) Average degree.



(d) Average weighted degree.

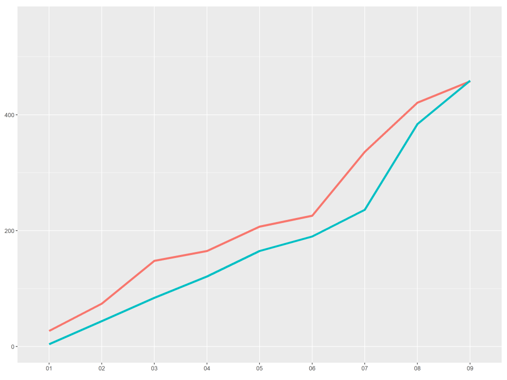


(e) Density.

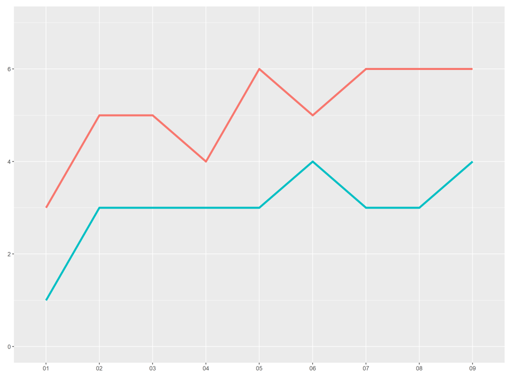


(f) Modularity.

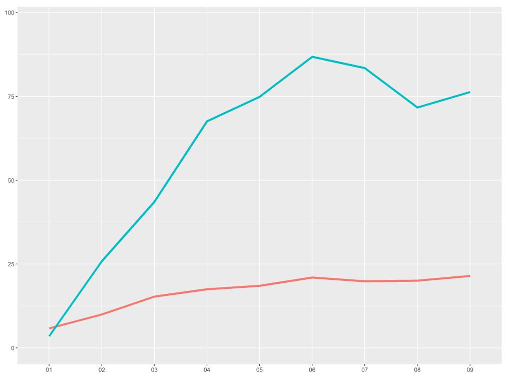
Figure 6.20.: Graph metrics of Zookeeper for simulation with refactorings. The empirical progress is colored red and the simulated progress is colored blue.



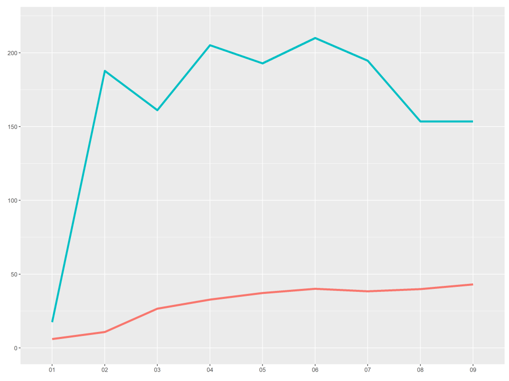
(a) Number of nodes (growth).



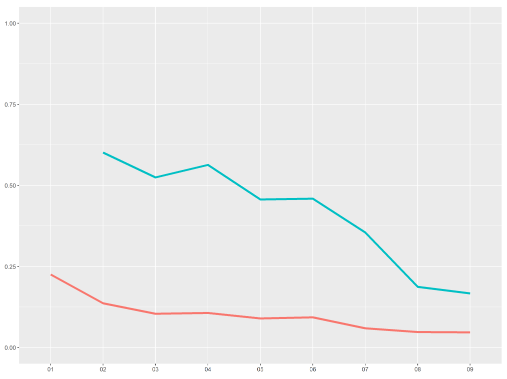
(b) Diameter.



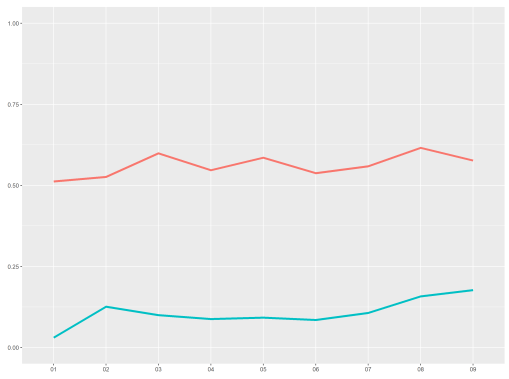
(c) Average degree.



(d) Average weighted degree.

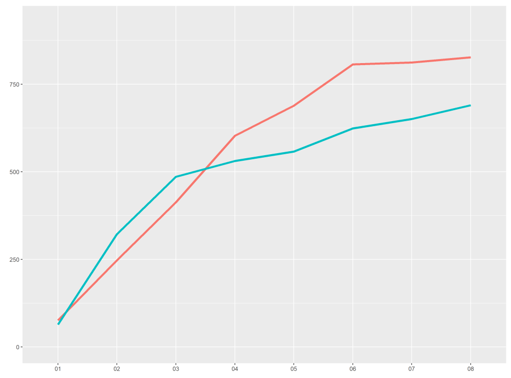


(e) Density.

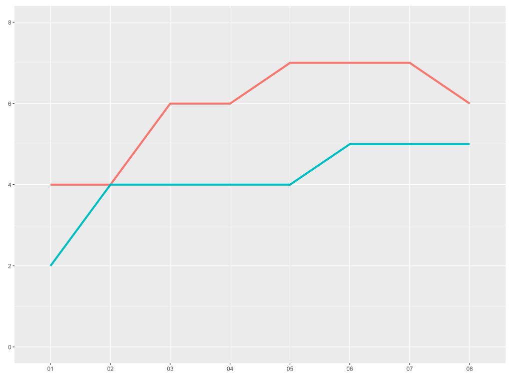


(f) Modularity.

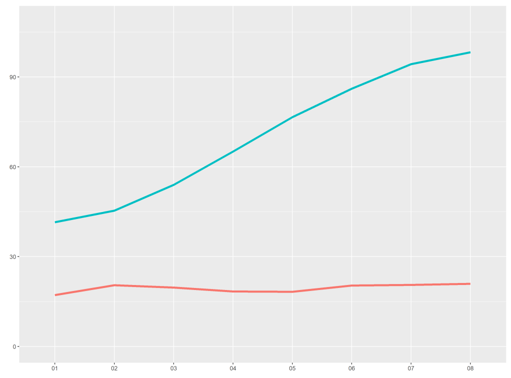
Figure 6.21.: Graph metrics of Gora for simulation with refactorings. The empirical progress is colored red and the simulated progress is colored blue.



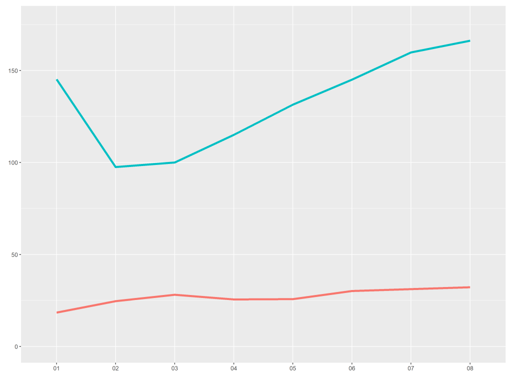
(a) Number of nodes (growth).



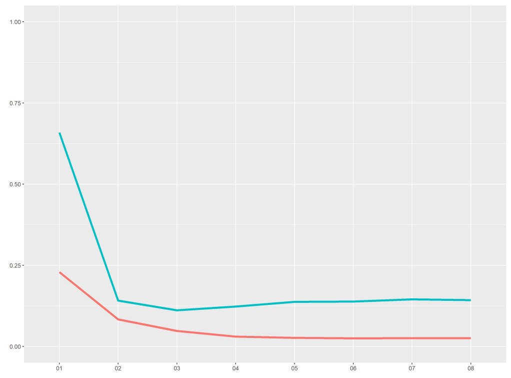
(b) Diameter.



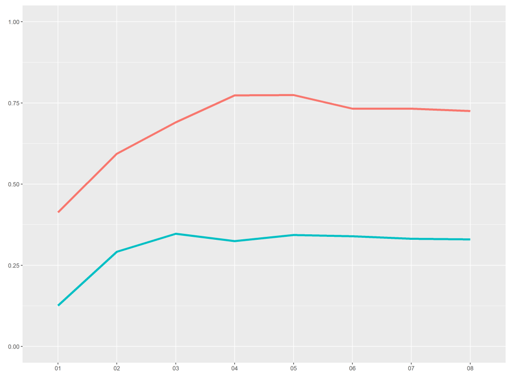
(c) Average degree.



(d) Average weighted degree.



(e) Density.



(f) Modularity.

Figure 6.22.: Graph metrics of Directory Fortress Core for simulation with refactorings. The empirical progress is colored red and the simulated progress is colored blue.

7. Discussion

Contents

In this chapter, the results of this thesis are discussed as a whole and research contributions as well as limitations of this work are presented.

The evolution of our proposed simulation model, in combination with the first case study, shows that we can determine the entities and relationships that are required to simulate several aspects of software evolution. The model without dependencies between the agents can reproduce the growth of a real software project when it is initialized with the contribution behavior of the developers as well as the maximum size and the duration of the project. This means that these are important parameters in the simulated software evolution process. To analyze more aspects of software evolution, we added more developer types, different bug types, and networks representing relationships between the entities to the simulation model. One of these networks represents the change coupling graph. As a representative of the software's semantic context, this graph is also used to analyze the quality of the simulated software. With such a model we can analyze aspects concerning the team constellation of the project. For example, the question what impact it has when a core developer leaves the project can be answered.

When we initialize the simulation model with project specific parameters generated by our mining tool, we are able to reproduce the common sub-linear growth trend as well as a linear and a super-linear growth trends of the analyzed project. Concerning the other compared graph metrics between simulated and real change coupling graph the results are split. In particular, the modularity and the average degree of the nodes differ simulative from the real values. This can be significantly improved if the simulation is started with real data after about one third of the project duration. This suggests that many software projects have an unpredictable behavior in their initial phase. For this reason, it is an important possibility to instantiate the simulation with real data at a certain point time to predict the future of a project.

Furthermore, we have shown that one can determine required software metrics and additional entities to model software refactorings using ABMS and graph transformations in combination. Using a model that only executes refactorings based on a given software snapshot, the application of refactorings works as expected and the quality of the software improves in terms of the complexity. However, the integration of this model into the model for software evolution only results in a slightly improved growth of the simulated project. We have expected this differently because the randomness in the file selection process for a commit is reduced by the transformation rules, but it can be explained by the small number of modeled refactorings and general commit patterns so far. It may also be that such an extended model is too complex to significantly improve the results. On the one hand, the substantial mining effort to model further transformation rules may not be justified if one looks at the current results concerning the simulated change coupling graph. On the other hand, the extension of the model would raise further questions regarding the impact of various refactoring strategies on the quality of the simulated software.

7.1. Contributions

The research contributions of this work are versatile and briefly presented in this section.

Showing that ABMS can be used to model software evolution. We presented an evolutionary ABM with the needed entities and dependencies between the entities to simulate software evolution. In terms of quality, we compare selected graph metrics of the simulated with the real change coupling graph.

Answering questions concerning aspects of software evolution with the ABM. We used our proposed ABM to answer specific questions concerning the reproducibility of growth or the impact of a leaving core developer. For the latter, the lifetime of bugs is implicitly needed.

Instantiate the ABM with project specific parameters. We developed an automated parameter estimation tool that provides project specific parameters to instantiate the proposed ABM. To generate these parameters, the different identities of a developer are merged, and developers are classified into different types according to the effort they spent. As a result of such instantiated models, we can reproduce several growth trends like sub-linear, linear, and super-liner ones.

Instantiate the ABM whit a snapshot of a real project. An output of automated parameter estimation tool is also the real change coupling graph of a given point in time. This graph is extended with developer information to instantiate the simulation model at this point. We showed that the metrics of the simulated change coupling graph of models instantiated after one third of the project duration fits the metrics of the real graph.

Mining parameters to model refactorings. We developed a mining framework that computes the changes of selected class and method metrics between two software revisions when a refactoring is applied. Based on this knowledge we implemented the SmartSHARK plug-in refSHARK that computes code entity states of changed classes and methods when a refactoring is applied in a commit. Both tools can output parameters that describe how the software changes on class and method level when a refactoring is executed. Furthermore, we developed a tool based on SartSHARK that classifies commits using a keyword-based classifier to get parameters of software changes on class and method level for commit types like bugfix or feature add.

Modeling refactorings using graph transformations. We present graph transformation rules with suitable parameters for changes on class and method level for the three most occurring refactorings in analyzed projects. That these rules work was shown on the basis of an example project, where after the initialization only refactorings were applied.

Integrating the refactoring model into software evolution model. We present a ABM extended by classes and methods as well as related metrics. This model is used to apply the transformation rules for refactorings and for commits like bugfixes or feature adds the class and method metrics change according to their commit pattern. With this model, we can slightly improve the growth trend of the simulated software.

7.2. Limitations

In this section, the limitations of our approach are presented.

Validation of the simulation results. Every simulation run with the same parameters outputs slightly different results due to the stochastic process that comes from the nature of simulation. In order to obtain comparable results, the simulation can be repeated several times with the same parameter set. Furthermore, the simulated results are compared with the results of the mining process. It cannot be determined whether the results of the mining process represent the real world project.

Runtime and memory consumption of the developed mining tools. A drawback of the developed mining tools, especially the change coupling graph generation, is the memory consumption. This is because we build up the graph successively and expand it with more and more data over time. We can analyze projects with about 6000 commits and 2000 files.

Runtime of the simulation. Using Repast Simphony for agent-based modeling and simulation purposes we can simulate projects with about 10000 files. For larger projects we have to consider other tools like the scalable simulation platform developed by our project partner [139].

Third party tools used for mining. Some of the mining tools used by the tools we have developed are not developed by us and we must rely on its results.

Metric selection. We use software metrics to describe and simulate several aspects of software evolution, for example, the number of files for the growth. Appropriate metrics are selected based on related work and achieved simulation results. For the modeling of the contribution behavior of the developers we only consider the commit frequency and the number of changes per commit. In [142] also the communication behavior is considered. Nevertheless, we think that our approach is suitable for the models presented in this thesis.

Project selection. The selection of projects to analyze is not only limited by its size according to the maximum size capable for the mining or simulation tool. It is also limited to the possibilities of the used mining framework. For the most mining work in this thesis we are using SmartSHARK. Therefore, we are limited to projects managed in git repositories. When this evaluation was made, the database hosted at the institute contained about 15 projects that meet all the requirements of our tools. From this we have selected six projects in such a way that we have at least two examples for each growth type.

8. Conclusions

Contents

In this chapter we summarize this thesis including our main findings and we present an outlook on possible future work.

8.1. Summary

This thesis introduces the evolution of an agent-based simulation model to simulate software processes as well as mining tools to gather suitable parameters for the instantiation of the model. This work is based on several research areas like software evolution, multiagent systems, refactorings, graph transformations, and mining software repositories. The interplay of these research areas makes the following statements possible.

We presented a simple ABM without dependencies and this model can reproduce the growth of a software system when it is initialized with parameters for the contribution behavior of the developers. By adding entities and dependencies to the model, more and more questions related to software evolution can be answered. One of these question is: can we simulate aspects like the loss of the core developer?

When we initialize the simulation model with project specific parameters generated by the developed automated parameter estimation tool in combination with minor modifications using the adjustable simulation parameters, we can reproduce the growth trend of the analyzed project. This means, we can simulate linear, sub-liner, and super-linear growth trends with such parametrized models. We figured out that the most accurate simulated change coupling graph according to all compared graph metrics is generated with a snapshot initialization of the simulation model. For this, we instantiate the model besides the base parameters with a change coupling graph of a given point in time. As a starting point, we selected about one third of the entire project duration.

Furthermore, a closer analysis of applied refactorings in open source repositories allows us to determine parameters to model software refactorings based on graph transformations. To integrate the rules into our previous agent-based model, we add new entities and dependencies representing an abstract software graph on method level. This model improves the simulated growth of a project slightly.

8.2. Main Findings

We have shown that the developed ABM presented in Section 4.2.3 is able to answer various questions of a project manager. For example, these questions can concern the lifetime of bugs or the team constellation based on the developer types working on the project. As a result, the simulation indicates different trends and no special numerical values for aspects like the system growth, the effort spent by the developers, or the number of open bugs.

Concerning the quality of the simulation we figured out that the ABM presented in Section 4.2.3 can reproduce common growth trends of real software projects. For this, the model is instantiated with project specific parameters (see Section 5.2). Furthermore, the comparison of graph metrics of the simulated and the real change coupling graph reveals that the results for density and diameter are significantly better than the trends for modularity and degree. The simulated graph can be improved by initializing the model based on a snapshot of the real project after about one third of the project duration. This improves the trends of all metrics, especially modularity and degree.

The integration of refactorings in our simulation model is far advanced but not conclusive. We figured out that the modeling of refactorings with graph transformations (see Section 4.3) works for the considered refactorings. Furthermore, we can parameterize the transformation rules based on our mining process described in Section 5.3. The present integration of the refactoring model (see Section 4.3) results in a slightly improved simulated growth trend. To be able to make further statements more transformation rules are required. These rules include further refactoring as well as other commits.

8.3. Outlook

The results of this work show that we can reproduce a realistically change coupling graph with our simulation model according to the growth and other compared graph metrics. Furthermore, the model can answer several questions concerning software evolution. To improve or to extend the model, the following work is planned.

To extend the number of possible projects to analyze we want to spend more effort in optimizing our mining and simulation tools. Another aim concerning all used tools is a common interface between SmarSHARK, our developed parameter estimation tools, and the simulation framework. Such an interface would allow a project manager to select a project to analyze on a GUI. For this project, automatically required parameters would be created and the manager could be shown various simulation results for different questions and parameter sets.

Concerning the modeled developers, we currently use a fixed number of developers over the entire project duration with an average change behavior per applied commit. It might be worthwhile to model the average change behavior per developer type. Furthermore, a close look at the project period and how many developers of a certain type are working on the project at a certain point could improve the results. This is especially true for the difficult to predict initial project phase.

For the bug generation, we use currently the bug introduction probabilities based on the ITS and bugs are created by the simulation context. Since the required metrics are already available from the simulated networks, we plan to generate bugs by certain commits which depend on the ownership as well as on the coupling degree of the entity which correlates with defects [137].

If the effort is made to extend the refactoring model, which means to model further refactorings and more general commit patterns, then further questions regarding software evolution can be investigated. For example, different refactoring strategies can then be investigated. Furthermore, when appropriate metrics are found to simulate maintainability, one can exploit in the simulation that classes of lower maintainability are subject to more refactorings [125].

In order to evaluate the refactoring model more accurate, we should be able to to compare the refactoring model with the model without refactorings initialized with a snapshot of the real software. For this, we must work on the retrieval of the snapshot of the extended software graph on class and method level including all considered software metrics.

Bibliography

[1] Reem Alfayez et al. “How does contributors involvement influence open source systems”. In: *2017 IEEE 28th Annual Software Technology Conference (STC)*. IEEE. 2017, pp. 1–8.

[2] Thomas Ball et al. “If your version control system could talk”. In: *ICSE Workshop on Process Modelling and Empirical Studies of Software Engineering*. 1997.

[3] Gerhard Weiss. *Multiagent Systems*. MIT Press, 2013.

[4] Verena Honsel, Daniel Honsel, and Jens Grabowski. “Software Process Simulation Based on Mining Software Repositories”. In: *ICDM Workshop*. 2014.

[5] Daniel Honsel et al. “Monitoring Software Quality by Means of Simulation Methods”. In: *10th International Symposium on Empirical Software Engineering and Measurement (ESEM)*. 2016.

[6] Martin Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.

[7] Daniel Honsel et al. “Simulating Software Refactorings Based on Graph Transformations”. In: *Simulation Science*. Springer. 2017, pp. 161–175.

[8] Verena Honsel et al. “Mining Software Dependency Networks for Agent-Based Simulation of Software Evolution”. In: *ASE Workshop*. 2015.

[9] V. Honsel. “Statistical Learning and Software Mining for Agent Based Simulation of Software Evolution”. In: *Doctoral Symposium at the 37th International Conference on Software Engineering (ICSE)*. 2015.

[10] Michael J. North et al. “Complex adaptive systems modeling with Repast Simphony”. English. In: *Complex Adaptive Systems Modeling* (2013).

[11] Michael Wooldridge and Nicholas R. Jennings. “Intelligent agents: Theory and practice”. In: *The knowledge engineering review* 10.2 (1995), pp. 115–152.

[12] Stuart J. Russell and Peter Norvig. *Artificial intelligence : a modern approach*. Prentice-Hall, 2010.

[13] RoboCup Technical Committee. *RoboCup Standard Platform League (Nao) Rule Book*. online. May 2011. URL: `http://spl.robocup.org/wp-content/uploads/downloads/Rules2011.pdf`.

[14] B-Human. *B-Human homepage.* 2019. URL: `https://b-human.de/index.html`.

[15] Hiroaki Kitano et al. *Robocup: The robot world cup initiative.* 1995.

[16] Thomas Röfer et al. *B-Human Team Report and Code Release 2011.* Only available online: `http://www.b-human.de/downloads/bhuman11_coderelease.pdf`. 2011.

[17] SoftBank. *SoftBank homepage.* 2019. URL: `https://www.softbankrobotics.com/emea/en/nao`.

[18] Seth Tisue and Uri Wilensky. "NetLogo: A simple environment for modeling complexity". In: *in International Conference on Complex Systems.* 2004, pp. 16–21.

[19] Arnaud Grignard et al. "GAMA 1.6: Advancing the Art of Complex Agent-Based Modeling and Simulation." In: *PRIMA*. Ed. by Guido Boella et al. Vol. 8291. Lecture Notes in Computer Science. Springer, 2013, pp. 117–131. URL: `http://dblp.uni-trier.de/db/conf/prima/prima2013.html#GrignardTGVHD13`.

[20] Steven F. Railsback, Steven L. Lytinen, and Stephen K. Jackson. "Agent-based simulation platforms: Review and development recommendations". In: *Simulation* 82.9 (2006), pp. 609–623.

[21] Charles M. Macal and Michael J. North. "Introductory tutorial: Agent-based modeling and simulation". In: *Simulation Conference (WSC), Proceedings of the 2011 Winter.* IEEE. 2011, pp. 1451–1464.

[22] Eclipse-Foundation. *The Platform for Open Innovation and Collaboration.* 2019. URL: `https://www.eclipse.org/`.

[23] Ross Ihaka and Robert Gentleman. "R: a language for data analysis and graphics". In: *Journal of computational and graphical statistics* 5.3 (1996), pp. 299–314.

[24] Mark Hall et al. "The WEKA data mining software: an update". In: *ACM SIGKDD explorations newsletter* 11.1 (2009), pp. 10–18.

[25] Brian Harvey. *Computer Science Logo Style: Symbolic Computing.* Vol. 1. MIT press, 1997.

[26] Jana Görmer et al. "JREP: Extending Repast Simphony for JADE Agent Behavior Components." In: *IAT*. Ed. by Olivier Boissier et al. IEEE Computer Society, 2011, pp. 149–154.

[27] Fabio Luigi Bellifemine, Giovanni Caire, and Dominic Greenwood. *Developing multi-agent systems with JADE.* Vol. 7. John Wiley & Sons, 2007.

[28] Serge Demeyer and Tom Mens. *Software Evolution.* Springer, 2008.

[29] Nazim H. Madhavji, Juan Fernandez-Ramil, and Dewayne Perry. *Software evolution and feedback: Theory and practice.* John Wiley & Sons, 2006.

[30] Winston W. Royce. "Managing the development of large software systems: concepts and techniques". In: *Proceedings of the 9th international conference on Software Engineering.* IEEE Computer Society Press (August 1970) – Reprinted in Proc. Int. Conf. Software Engineering (ICSE) 1989. 1989, pp. 328–338.

[31] IEEE. *Standard IEEE Std 1219-1999 on Software Maintenance.* IEEE Press. Volume 2. 1999.

[32] Meir M Lehman. "On understanding laws, evolution, and conservation in the large-program life cycle". In: *Journal of Systems and Software* 1 (1979), pp. 213–221.

[33] Meir M. Lehman. "Programs, life cycles, and laws of software evolution". In: *Proc. IEEE* 68.9 (Sept. 1980), pp. 1060–1076.

[34] Manny M Lehman and Laszlo A Belady. *Program evolution: processes of software change.* Academic Press Professional, Inc., 1985.

[35] Meir M Lehman et al. "Metrics and laws of software evolution – The Nineties View". In: *Proceedings Fourth International Software Metrics Symposium.* IEEE. 1997, pp. 20–32.

[36] M.M. Lehman and J.F. Ramil. "Towards a Theory of Software Evolution - And its practical impact (working paper)". In: *Invited Talk, Proceedings Intl. Symposium on Principles of Softw. Evolution, ISPSE 2000, 1-2 Nov.* Press, 2000, pp. 2–11.

[37] William Thomson. "Electrical units of measurement". In: *Popular lectures and addresses* 1.73 (1883).

[38] Jochen Ludewig and Horst Lichter. *Software Engineering: Grundlagen, Menschen, Prozesse, Techniken.* dpunkt. verlag, 2013.

[39] IEEE. *IEEE Standard Glossery of Software Engineering Terminology.* IEEE Press. 1990.

[40] Victor R Basili and David M Weiss. "A methodology for collecting valid software engineering data". In: *IEEE Transactions on software engineering* 6 (1984), pp. 728–738.

[41] Norman Fenton and James Bieman. *Software metrics: a rigorous and practical approach.* CRC press, 2014.

[42] Shyam R Chidamber and Chris F Kemerer. "A metrics suite for object oriented design". In: *IEEE Transactions on software engineering* 20.6 (1994), pp. 476–493.

[43] M. Fowler. *Refactoring.* 2019. URL: `https://refactoring.com/`.

[44] Kyle Prete et al. "Template-based reconstruction of complex refactorings". In: *2010 IEEE International Conference on Software Maintenance.* IEEE. 2010, pp. 1–10.

[45] Danilo Silva and Marco Tulio Valente. “RefDiff: detecting refactorings in version histories”. In: *Proceedings of the 14th International Conference on Mining Software Repositories*. IEEE Press. 2017, pp. 269–279.

[46] Grzegorz Rozenberg. *Handbook of Graph Grammars and Computing by Graph Transformation*. Vol. 1. World scientific, 1997.

[47] Rozenberg Grzegorz. *Handbook Of Graph Grammars And Computing By Graph Transformations, Vol 2: Applications, Languages And Tools*. world Scientific, 1999.

[48] Hartmut Ehrig, Grzegorz Rozenberg, and Hans-J rg Kreowski. *Handbook of graph grammars and computing by graph transformation*. Vol. 3. world Scientific, 1999.

[49] James Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified modeling language reference manual, the*. Pearson Higher Education, 2004.

[50] Peter Pin-Shan Chen. “The entity-relationship model – toward a unified view of data”. In: *ACM Transactions on Database Systems (TODS)* 1.1 (1976), pp. 9–36.

[51] Hans-Jörg Kreowski, Renate Klempien-Hinrichs, and Sabine Kuske. “Some Essentials of Graph Transformation”. In: *Recent advances in formal languages and applications* 25 (2006), pp. 229–254.

[52] Andrea Corradini et al. “Algebraic approaches to graph transformation–part i: Basic concepts and double pushout approach”. In: *Handbook Of Graph Grammars And Computing By Graph Transformation: Volume 1: Foundations*. World Scientific, 1997, pp. 163–245.

[53] Richard J Trudeau. *Introduction to graph theory*. Courier Corporation, 2013.

[54] Maarten Van Steen. “Graph theory and complex networks”. In: *An introduction* 144 (2010).

[55] Tore Opsahl, Filip Agneessens, and John Skvoretz. “Node centrality in weighted networks: Generalizing degree and shortest paths”. In: *Social Networks* 32.3 (2010), pp. 245–251.

[56] Mathieu Bastian, Sebastien Heymann, Mathieu Jacomy, et al. “Gephi: an open source software for exploring and manipulating networks.” In: *Proc. of the 3rd Intern. AAAI Conf. on Weblogs and Social Media (ICWSM)*. 2009.

[57] Santo Fortunato. “Community detection in graphs”. In: *Physics Reports* 486.3-5 (2010), pp. 75–174. DOI: DOI:10.1016/j.physrep.2009.11.002.

[58] Jure Leskovec, Kevin J Lang, and Michael Mahoney. “Empirical comparison of algorithms for network community detection”. In: *Proceedings of the 19th international conference on World wide web*. ACM. 2010, pp. 631–640.

[59] Vincent D Blondel et al. “Fast unfolding of communities in large networks”. In: *Journal of statistical mechanics: theory and experiment* 2008.10 (2008), P10008.

[60] Paolo Bottoni, Francesco Parisi-Presicce, and Gabriele Taentzer. "Specifying Coherent Refactoring Software Artefacts with Distributed Graph Transformations". In: *Transformation of Knowledge, Information and Data: Theory and Applications*. IGI Global, 2005, pp. 95–126.

[61] Tom Mens et al. "Formalizing refactorings with graph transformations". In: *Journal of Software Maintenance and Evolution: Research and Practice* 17.4 (2005), pp. 247–276.

[62] Hartmut Ehrig et al. "Algebraic approaches to graph transformation–part II: Single pushout approach and comparison with double pushout approach". In: *Handbook Of Graph Grammars And Computing By Graph Transformation: Volume 1: Foundations*. World Scientific, 1997, pp. 247–312.

[63] Hadi Hemmati et al. "The msr cookbook: Mining a decade of research". In: *Proceedings of the 10th Working Conference on Mining Software Repositories*. IEEE Press. 2013, pp. 343–352.

[64] Marco D'Ambros et al. "Analysing Software Repositories to Understand Software Evolution". English. In: *Software Evolution*. Springer Berlin Heidelberg, 2008, pp. 37–67.

[65] Adrian Bachmann et al. "The missing links: bugs and bug-fix commits." In: *SIGSOFT FSE*. Ed. by Gruia-Catalin Roman and Kevin J. Sullivan. ACM, 2010, pp. 97–106.

[66] Nicolas Bettenburg, Emad Shihab, and Ahmed E. Hassan. "An empirical study on the risks of using off-the-shelf techniques for processing mailing list data." In: *ICSM*. IEEE Computer Society, 2009, pp. 539–542.

[67] Huzefa Kagdi, Michael L Collard, and Jonathan I Maletic. "Towards a taxonomy of approaches for mining of source code repositories". In: *ACM SIGSOFT Software Engineering Notes*. Vol. 30. 4. ACM. 2005, pp. 1–5.

[68] Kamel Ayari et al. "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*. IBM Corp. 2007, pp. 215–228.

[69] Mathieu Goeminne and Tom Mens. "A comparison of identity merge algorithms for software repositories". In: *Science of Computer Programming* 78.8 (2013), pp. 971–986.

[70] Apache. *Subversion homepage*. 2019. URL: `https://subversion.apache.org/`.

[71] Microsoft. *Azure DevOps Server homepage*. 2019. URL: `https://azure.microsoft.com/en-us/services/devops/server/`.

[72] *git homepage*. 2019. URL: `https://git-scm.com/`.

[73] *Mercurial homepage*. 2019. URL: `https://www.mercurial-scm.org/`.

[74] Christian Bird et al. “The promises and perils of mining git”. In: *2009 6th IEEE International Working Conference on Mining Software Repositories.* IEEE. 2009, pp. 1–10.

[75] Terry Weissman et al. *Bugzilla.* http://www.bugzilla.org. 1998. URL: `http://www.bugzilla.org`.

[76] Atlassian. *Jira homepage.* 2019. URL: `https://www.atlassian.com/software/jira`.

[77] FrontEndART. *SourceMeter – Homepage.* Online. URL: `https://www.sourcemeter.com/`.

[78] Robert Dyer et al. “Boa: A language and infrastructure for analyzing ultra-large-scale software repositories”. In: *Proceedings of the 2013 International Conference on Software Engineering.* IEEE Press. 2013, pp. 422–431.

[79] Fabian Trautsch et al. “Addressing problems with replicability and validity of repository mining studies through a smart data platform”. In: *Empirical Software Engineering* 23.2 (2018), pp. 1036–1083.

[80] Neil Smith and Juan Fernández Ramil. “Agent-based simulation of open source evolution”. In: *Software Process Improvement and Practice.* 2006.

[81] Marc I Kellner, Raymond J Madachy, and David M Raffo. “Software process simulation modeling: why? what? how?” In: *Journal of Systems and Software* 46.2-3 (1999), pp. 91–105.

[82] He Zhang, Barbara Kitchenham, and Dietmar Pfahl. “Software process simulation modeling: an extended systematic review”. In: *International Conference on Software Process.* Springer. 2010, pp. 309–320.

[83] Redha Cherif and Paul Davidsson. “Software development process simulation: multi agent-based simulation versus system dynamics”. In: *International Workshop on Multi-Agent Systems and Agent-Based Simulation.* Springer. 2009, pp. 73–85.

[84] Yongqin Gao and Greg Madey. “Towards Understanding: A Study of the SourceForge.Net Community Using Modeling and Simulation”. In: *Proceedings of the 2007 Spring Simulation Multiconference - Volume 2.* SpringSim ’07. Norfolk, Virginia: Society for Computer Simulation International, 2007, pp. 145–150.

[85] Nelson Minar et al. “The swarm simulation system: A toolkit for building multi-agent simulations”. In: (1996).

[86] Bojan Spasic and Bhakti S. S. Onggo. “Agent-based simulation of the software development process: a case study at AVL.” In: *Winter Simulation Conference.* Ed. by Oliver Rose and Adelinde M. Uhrmacher. WSC, 2012, 400:1–400:11. URL: `http://dblp.uni-trier.de/db/conf/wsc/wsc2012.html#SpasicO12`.

[87] Mary Beth Chrissis, Mike Konrad, and Sandra Shrum. *CMMI for development: guidelines for process integration and product improvement.* Pearson Education, 2011.

[88] Ravikant Agarwal and David Umphress. "A flexible model for simulation of software development process". In: *Proceedings of the 48th Annual Southeast Regional Conference.* ACM. 2010, p. 40.

[89] Wladyslaw M Turski. "Reference model for smooth growth of software systems". In: *IEEE Transactions on Software Engineering* 8 (1996), pp. 599–600.

[90] Michael W. Godfrey and Qiang Tu. "Evolution in Open Source Software: A Case Study". In: *Proc. Int'l Conf. Software Maintenance (ICSM).* Los Alamitos, California: IEEE Computer Society Press, 2000, pp. 131–142.

[91] James W Paulson, Giancarlo Succi, and Armin Eberlein. "An empirical study of open-source and closed-source software products". In: *IEEE Transactions on Software Engineering* 30.4 (2004), pp. 246–256.

[92] Herraiz Israel and Robles Gregorio. "Comparison between SLOCs and number of files as size metrics for software evolution analysis". In: *Washington, USA: Proceedings of the Conference on Software Maintenance and R-eengineering. IEEE Computer Society.* 2006, pp. 206–213.

[93] G. Robles et al. "Evolution and growth in large libre software projects". In: *Author Index* (2005), pp. 165–174.

[94] Andrea Capiluppi and Juan F Ramil. "Studying the evolution of open source systems at different levels of granularity: Two case studies". In: *Proceedings. 7th International Workshop on Principles of Software Evolution, 2004.* IEEE. 2004, pp. 113–118.

[95] Marco D'Ambros, Michele Lanza, and Romain Robbes. "On the Relationship Between Change Coupling and Software Defects". In: *Proc. of the 16th Working Conf. on Rev. Eng.* IEEE Computer Society, 2009.

[96] Chakkrit Tantithamthavorn, Akinori Ihara, and Ken-ichi Matsumoto. "Using co-change histories to improve bug localization performance". In: *2013 14th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing.* IEEE. 2013, pp. 543–548.

[97] Igor Scaliante Wiese et al. "An empirical study of the relation between strong change coupling and defects using history and social metrics in the apache aries project". In: *IFIP International Conference on Open Source Systems.* Springer. 2015, pp. 3–12.

[98] John Anvik, Lyndon Hiew, and Gail C. Murphy. "Who Should Fix This Bug?" In: *Proceedings of the 28th International Conference on Software Engineering.* ICSE '06. Shanghai, China: ACM, 2006, pp. 361–370. DOI: `10.1145/1134285.1134336`. URL: `http://doi.acm.org/10.1145/1134285.1134336`.

[99] Sunghun Kim, E. J. Whitehead, and Yi Zhang. “Classifying Software Changes: Clean or Buggy?” In: *Software Engineering, IEEE Transactions on* (2008).

[100] Ahmed Lamkanfi et al. “Predicting the severity of a reported bug”. In: *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010).* IEEE. 2010, pp. 1–10.

[101] Cathrin Weiss et al. “How long will it take to fix this bug?” In: *Fourth International Workshop on Mining Software Repositories (MSR’07: ICSE Workshops 2007).* IEEE. 2007, pp. 1–1.

[102] Liguo Yu and Srini Ramaswamy. “Mining CVS Repositories to Understand Open-Source Project Developer Roles”. In: *Proceedings of the Fourth International Workshop on Mining Software Repositories.* 2007.

[103] Kevin Crowston et al. “Core and periphery in free/libre and open source software team communications”. In: *Proceedings of the 39th Annual Hawaii International Conference on System Sciences (HICSS’06).* Vol. 6. IEEE. 2006, 118a–118a.

[104] Antonio Terceiro, Luiz Romario Rios, and Christina Chavez. “An empirical study on the structural complexity introduced by core and peripheral developers in free software projects”. In: *2010 Brazilian Symposium on Software Engineering.* IEEE. 2010, pp. 21–29.

[105] Chintan Amrit and Jos Van Hillegersberg. “Exploring the Impact of Soclo-Technlcal Core-Periphery Structures in Open Source Software Development”. In: *journal of information technology* 25.2 (2010), pp. 216–229.

[106] Mitchell Joblin et al. “Classifying developers into core and peripheral: An empirical study on count and network metrics”. In: *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE).* IEEE. 2017, pp. 164–174.

[107] Kevin Crowston and James Howison. “The social structure of free and open source software development”. In: *First Monday* 10.2 (2005).

[108] Gregorio Robles, Jesus M Gonzalez-Barahona, and Israel Herraiz. “Evolution of the core team of developers in libre software projects”. In: *2009 6th IEEE international working conference on mining software repositories.* IEEE. 2009, pp. 167–170.

[109] Pamela Bhattacharya et al. “Graph-based Analysis and Prediction for Software Evolution”. In: *Proceedings of the 34th Intern.Conf. on Softw. Eng. (ICSE).* Zurich, Switzerland: IEEE, 2012.

[110] Xu Ben, Shen Beijun, and Yang Weicheng. “Mining Developer Contribution in Open Source Software Using Visualization Techniques”. In: *Proceedings of the Third International Conference on Intelligent System Design and Engineering Applications (ISDEA).* 2013, pp. 934–937. DOI: `10.1109/ISDEA.2012.223`.

[111] Matthieu Foucault et al. “Impact of Developer Turnover on Quality in Open-source Software”. In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering.* Bergamo, Italy, 2015.

[112] Jalerson Lima et al. “Assessing developer contribution with repository mining-based metrics”. In: *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME).* IEEE. 2015, pp. 536–540.

[113] Georgios Gousios, Eirini Kalliamvakou, and Diomidis Spinellis. “Measuring Developer Contribution from Software Repository Data”. In: *Proceedings of the 2008 International Working Conference on Mining Software Repositories.* Leipzig, Germany, 2008.

[114] Tudor Girba et al. “How developers drive software evolution”. In: *Eighth International Workshop on Principles of Software Evolution (IWPSE’05).* IEEE. 2005, pp. 113–122.

[115] Philip Makledonski. “Developer-Centric Software Assessment”. PhD thesis. University of Göttingen, June 2018.

[116] Lile Hattori and Michele Lanza. “On the nature of commits.” In: *ASE Workshops.* IEEE, 2008, pp. 63–71.

[117] Peter Weissgerber and Stephan Diehl. “Identifying Refactorings from Source-Code Changes”. In: *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering.* ASE ’06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 231–240. DOI: `10.1109/ASE.2006.41`. URL: `http://dx.doi.org/10.1109/ASE.2006.41`.

[118] Deepak Advani, Youssef Hassoun, and Steve Counsell. “Extracting refactoring trends from open-source software and a possible solution to the’related refactoring’conundrum”. In: *Proceedings of the 2006 ACM symposium on Applied computing.* ACM. 2006, pp. 1713–1720.

[119] Nikolaos Tsantalis and Alexander Chatzigeorgiou. “Identification of move method refactoring opportunities”. In: *IEEE Transactions on Software Engineering* 35.3 (2009), pp. 347–367.

[120] Nikolaos Tsantalis et al. “Accurate and efficient refactoring detection in commit history”. In: *Proceedings of the 40th International Conference on Software Engineering.* ACM. 2018, pp. 483–494.

[121] Beat Fluri et al. “Change distilling: Tree differencing for fine-grained source code change extraction”. In: *IEEE Transactions on software engineering* 33.11 (2007), pp. 725–743.

[122] Jean-Rémy Falleri et al. “Fine-grained and Accurate Source Code Differencing”. In: *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering.* ASE ’14. Vasteras, Sweden: ACM, 2014, pp. 313–324. DOI: `10.1145/2642937.2642982`. URL: `http://doi.acm.org/10.1145/2642937.2642982`.

[123] *CVSAnalY on GitHub*. 2019. URL: `https://github.com/MetricsGrimoire/CVSAnalY`.

[124] Gregorio Robles et al. "Remote analysis and measurement of libre software systems by means of the CVSAnalY tool". In: *Proceedings of the 2nd ICSE Workshop on Remote Analysis and Measurement of Software Systems (RAMSS)*. IET. 2004, pp. 51–56.

[125] István Kádár et al. "A code refactoring dataset and its assessment regarding software maintainability". In: *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. Vol. 1. IEEE. 2016, pp. 599–603.

[126] István Kádár et al. "Assessment of the Code Refactoring Dataset Regarding the Maintainability of Methods". In: *International Conference on Computational Science and Its Applications*. Springer. 2016, pp. 610–624.

[127] Tom Mens. "Conditional graph rewriting as a domain-independent formalism for software evolution". In: *International Workshop on Applications of Graph Transformations with Industrial Relevance*. Springer. 1999, pp. 127–143.

[128] Tom Mens. "Transformational software evolution by assertions". In: *CSMR Workshop on Formal Foundations of Software Evolution, Lisbon*. 2001.

[129] Tom Mens, Serge Demeyer, and Dirk Janssens. "Formalising behaviour preserving program transformations". In: *International Conference on Graph Transformation*. Springer. 2002, pp. 286–301.

[130] G Taentzer et al. "Visual design of distributed systems by graph transformation". In: *Handbook of Graph Grammars and Computing by Graph Transformation* 3 (1999), pp. 269–340.

[131] Hans-Jörg Kreowski, Sabine Kuske, and Andy Schürr. "Nested graph transformation units". In: *International Journal of Software Engineering and Knowledge Engineering* 7.04 (1997), pp. 479–502.

[132] Michael J. North and Charles M. Macal. "Product Design Patterns for Agent-based Modeling". In: *Proceedings of the Winter Simulation Conference*. WSC '11. Phoenix, Arizona: Winter Simulation Conference, 2011, pp. 3087–3098.

[133] Tommaso Toffoli and Norman Margolus. *Cellular automata machines: a new environment for modeling*. MIT press, 1987.

[134] Eric Weisstein. *Moore Neighborhood*. Online. 2019. URL: `http://mathworld.wolfram.com/MooreNeighborhood.html`.

[135] Verena Honsel et al. "Developer Oriented and Quality Assurance Based Simulation of Software Processes". In: *Proceedings of the Seminar Series on Advanced Techniques & Tools for Software Evolution (SATToSE) 2015*. July 2015.

[136] Nachiappan Nagappan, Brendan Murphy, and Victor Basili. “The Influence of Organizational Structure on Software Quality: An Empirical Case Study”. In: *Proceedings of the 30th International Conference on Software Engineering (ICSE)*. Leipzig, Germany: ACM, 2008.

[137] Foyzur Rahman and Premkumar Devanbu. “Ownership, Experience and Defects: A Fine-grained Study of Authorship”. In: *Proc. of the 33rd Intern. Conf. on Softw. Eng. (ICSE)*. Waikiki, Honolulu, HI, USA, 2011.

[138] M. Ali and M. O. Elish. “A Comparative Literature Survey of Design Patterns Impact on Software Quality”. In: *2013 International Conference on Information Science and Applications (ICISA)*. June 2013, pp. 1–7. DOI: `10.1109/ICISA.2013.6579460`.

[139] Tobias Ahlbrecht, Jürgen Dix, and Niklas Fiekas. “Scalable Multi-Agent Simulation based on MapReduce”. In: *Proceedings of the 14th European Conference on Multi-Agent Systems*. EUMAS 2016. Springer, Dec. 2016.

[140] Tobias Ahlbrecht et al. “Agent-based simulation for software development processes”. In: *Proceedings of the 14th European Conference on Multi-Agent Systems*. EUMAS 2016. Springer, Dec. 2016.

[141] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. “When do changes induce fixes?” In: *ACM sigsoft software engineering notes*. Vol. 30. 4. ACM. 2005, pp. 1–5.

[142] Verena Herbold. “Mining Developer Dynamics for Agent-Based Simulation of Software Evolution”. PhD thesis. University of Göttingen, 2019.

[143] *Morphia – The JVM Object Document Mapper for MongoDB*. online. 2019.

[144] *MongoDB – Homepage*. online. URL: `https://www.mongodb.com/`.

Acronyms

ABM Agent-Based Model.

ABMS Agent-Based Modeling and Simulation.

AST abstract syntax tree.

BDI belief-desire-intention.

DES Descrete Event Simulation.

DPO double-pushout approach.

GIS Geographic Information System.

GUI Graphical User Interface.

IDE Integrated Development Environment.

ITS Issue Tracking System.

JADE JAVA Agent DEvelopment.

LOC Lines of Code.

McCC McCabe's Cyclomatic Complexity.

ML Mailing List.

MSR Mining Software Repositories.

NII Number of Incoming Invocations.

NOI Number of Outgoing Invocations.

OSS Open Source Software.

POJO Plain Old Java Object.

PSP Personal Software Process.

SD System Dynamics.

SPO single-pushout approach.

UML Unified Modeling Language.

VCS Version Control System.

WMC Weighted Methods per Class.

Glossary

agent

Autonomous entity acting based on its own behavior and gathered data about the (local) environment. 6, 10–13, 15, 16, 30, 31, 37, 38, 41, 42, 44–46, 54, 55, 69–71, 74, 93, 96–98

git

Distributed version control system. 22, 26, 58, 64, 65, 96

metric

A quantified statement about a product or a software process. 19–22, 25, 27, 31, 33–35, 51, 53, 55, 56, 59, 63–65, 132, 133

object

Compututional entity that ecaplulates a state (values of attributes), this state can be changed by method calls. 12, 16, 20, 22

refactoring

Technique for restructuring an existing body of code, altering its internal structure without changing its external behavior. 2, 3, 6, 10, 20–22, 26, 27, 29, 33–35, 37, 39–41, 47, 51–57, 63–65, 67, 69, 76, 82, 86, 87, 94, 95, 97–99, 117, 133–135, 138

List of Figures

List of Tables

A. Simulation Parameters from Mining

Contents

This appendix presents simulation parameters gathered by mining open source repositories as described on Chapter 5. For each analyzed project an own parameter set is created.

A.1. Core Parameter for Simulation Instantiation

The parameters presented below are the core parameters to initialize the simulation model. They are generated for each project by our automated parameter estimation tool described in Section 5.2. As an example, the parameters for the open source project oisafe[1] are shown below.

The basic data for each project are the maximum size of the project, the number and change probabilities of commits, the number of rounds to simulate, and the developers (identities) to instantiate with their role specific data. Furthermore,

[1]https://github.com/openintents/safe.git

information about bugs, their fixes, and the categories of a project are available. If the mining process reveals two different project phases such as one initial phase with high project growth and a following development phase with lower growth, both phases can be described with different probabilities for commit activities.

```
{
  "maxFiles": 73,
  "numberOfCommits": 233,
  "pAverageCommitUpdate": 0.4102112676056338,
  "pAverageCommitDelete": 0.9549180327868851,
  "pAverageCommitAdd": 0.7420382165605096,
  "numberOfInitialCommits": 0,
  "pInitialCommitUpdate": 0,
  "pInitialCommitDelete": 0,
  "pInitialCommitAdd": 0,
  "numberOfDevelopmentCommits": 233,
  "pDevelopmentCommitUpdate": 0.4102112676056338,
  "pDevelopmentCommitDelete": 0.9549180327868851,
  "pDevelopmentCommitAdd": 0.7420382165605096,
  "firstCommitDate": 1294515834000,
  "lastCommitDate": 1499363492000,
  "monthToSimulate": 78,
  "roundsToSimulate": 2372,
  "keyDeveloper": 3,
  "keyDeveloperCommits": 226,
  "keyDeveloperFixes": 6,
  "keyDeveloperMaintainer": 1,
  "majorDeveloper": 0,
  "majorDeveloperCommits": 0,
  "majorDeveloperFixes": 0,
  "majorDeveloperMaintainer": 0,
  "minorDeveloper": 4,
  "minorDeveloperCommits": 7,
  "minorDeveloperFixes": 1,
  "minorDeveloperMaintainer": 0,
  "peripheralDeveloper": 4,
  "peripheralDeveloperCommits": 7,
  "peripheralDeveloperFixes": 1,
  "peripheralDeveloperMaintainer": 0,
  "coreDeveloper": 3,
  "coreDeveloperCommits": 226,
  "coreDeveloperFixes": 6,
  "coreDeveloperMaintainer": 1,
"issueInformationComplete": {
  "CRITICAL": 0,
  "MINOR": 0,
  "MAJOR": 0,
  "NONE": 24
},
"issueInformationCompleteFixed": {
  "CRITICAL": 0,
  "MINOR": 0,
  "MAJOR": 0,
  "NONE": 13
},
"issueInformationYearly": {
  "2016": {
    "CRITICAL": 0,
    "MINOR": 0,
    "MAJOR": 0,
    "NONE": 5
  },
  "2012": {
    "CRITICAL": 0,
    "MINOR": 0,
    "MAJOR": 0,
    "NONE": 6
  },
  "2013": {
    "CRITICAL": 0,
    "MINOR": 0,
    "MAJOR": 0,
    "NONE": 5
  },
  "2014": {
    "CRITICAL": 0,
    "MINOR": 0,
    "MAJOR": 0,
    "NONE": 2
  },
  "2015": {
    "CRITICAL": 0,
    "MINOR": 0,
    "MAJOR": 0,
    "NONE": 6
  }
},
"exportPackages": [
  {
    "name": "org.openintents.safe",
    "files": 51,
    "percent": 79.6875
  },
  {
    "name": "org.openintents.safe.dialog",
    "files": 5,
    "percent": 7.8125
  },
  {
    "name": "org.openintents.safe.wrappers",
    "files": 4,
    "percent": 6.25
  },
  {
    "name": "org.openintents.safe.service",
    "files": 4,
    "percent": 6.25
  }
],
"identities": [
  {
    "objectID": "899b887e-8f0a-463d-a1ad-13573790a6b5",
    "name": "Peli",
    "numberOfCommits": 92,
    "percent": 39.48497854077253,
    "type": "key",
    "role": "core",
    "maintainer": false,
    "numberOfFixes": 0,
    "numberOfTests": 0,
    "numberOfFeatures": 0,
    "numberOfMaintenance": 0,
    "numberOfRefactorings": 0,
    "numberOfDocumentation": 0
  },
  {
    "objectID": "9bc989d5-9003-4d56-a8b7-3e99232022d6",
    "name": "Randy McEoin",
    "numberOfCommits": 75,
    "percent": 32.18884120171674,
    "type": "key",
    "role": "core",
    "maintainer": false,
    "numberOfFixes": 1,
    "numberOfTests": 0,
    "numberOfFeatures": 0,
    "numberOfMaintenance": 0,
    "numberOfRefactorings": 0,
    "numberOfDocumentation": 0
  },
  {
    "objectID": "dcb22e5f-3750-4aa2-9a28-4c6e2f6a4ac6",
    "name": "Friedger Mueffke",
    "numberOfCommits": 59,
    "percent": 25.321888412017167,
    "type": "key",
    "role": "core",
    "maintainer": true,
    "numberOfFixes": 5,
    "numberOfTests": 0,
    "numberOfFeatures": 0,
    "numberOfMaintenance": 0,
    "numberOfRefactorings": 0,
    "numberOfDocumentation": 0
  },
  {
    "objectID": "1941cc93-7186-447b-981d-ebde4517a903",
    "name": "openintents-bot",
    "numberOfCommits": 4,
    "percent": 1.7167381974248928,
    "type": "minor",
    "role": "peripheral",
    "maintainer": false,
    "numberOfFixes": 0,
    "numberOfTests": 0,
    "numberOfFeatures": 0,
    "numberOfMaintenance": 0,
    "numberOfRefactorings": 0,
    "numberOfDocumentation": 0
  },
  {
    "objectID": "05496fe2-978c-4fb9-9ffa-5e3b421f47f2",
    "name": "harshadura",
    "numberOfCommits": 1,
    "percent": 0.4291845493562232,
      "type": "minor",
      "role": "peripheral",
      "maintainer": false,
      "numberOfFixes": 0,
      "numberOfTests": 0,
      "numberOfFeatures": 0,
      "numberOfMaintenance": 0,
      "numberOfRefactorings": 0,
      "numberOfDocumentation": 0
    },
    {
      "objectID": "f398a65f-a7c4-478d-a13a-7541a563cd00",
      "name": "chaitanya",
      "numberOfCommits": 1,
      "percent": 0.4291845493562232,
      "type": "minor",
      "role": "peripheral",
      "maintainer": false,
      "numberOfFixes": 1,
      "numberOfTests": 0,
      "numberOfFeatures": 0,
      "numberOfMaintenance": 0,
      "numberOfRefactorings": 0,
      "numberOfDocumentation": 0
    },
    {
      "objectID": "e06e9142-6543-450f-9edd-6e176ca76bfa",
      "name": "andrew-codechimp",
      "numberOfCommits": 1,
      "percent": 0.4291845493562232,
      "type": "minor",
      "role": "peripheral",
      "maintainer": false,
      "numberOfFixes": 0,
      "numberOfTests": 0,
      "numberOfFeatures": 0,
      "numberOfMaintenance": 0,
      "numberOfRefactorings": 0,
      "numberOfDocumentation": 0
    }
  ]
}
```

A.2. Change Coupling Graph to Initialize the Simulation at Different Starting Points

The change coupling graph is generated by our automated parameter estimation tool like described in Section 5.2.3. As file format we use the dot format[2]. The nodes represent the files of the software and the edges with their weights represent how often files are changed together in one commit. To initialize the simulation, the nodes contain additional information like the owner, the creator, all developers who touched the file and how often they touched it, and the package the file belongs to. By default, it is generated for each year, but it is also possible to get the graph for any specific commit.

As an example, the change coupling graph of the the first analyzed year of the open source project oisafe[3] is shown below.

```
strict graph G {
  1 [ label="SafeDemo/src/org/openintents/intents/CryptoIntents.java"
     owner="899b887e-8f0a-463d-a1ad-13573790a6b5"
     creator="899b887e-8f0a-463d-a1ad-13573790a6b5"
     dev1="899b887e-8f0a-463d-a1ad-13573790a6b5;1"
     package="org.openintents.intents" ];
  2 [ label="SafeDemo/src/org/openintents/samples/testsafe/
         TestSafe.java"
     owner="899b887e-8f0a-463d-a1ad-13573790a6b5"
     creator="899b887e-8f0a-463d-a1ad-13573790a6b5"
     dev1="899b887e-8f0a-463d-a1ad-13573790a6b5;1"
     package="org.openintents.samples.testsafe" ];
  3 [ label="Safe/src/org/openintents/safe/dialog/
         DialogHostingActivity.java"
     owner="899b887e-8f0a-463d-a1ad-13573790a6b5"
     creator="899b887e-8f0a-463d-a1ad-13573790a6b5"
     dev1="899b887e-8f0a-463d-a1ad-13573790a6b5;2"
     package="org.openintents.safe.dialog" ];
  4 [ label="Safe/src/org/openintents/safe/AskPassword.java"
     owner="9bc989d5-9003-4d56-a8b7-3e99232022d6"
     creator="899b887e-8f0a-463d-a1ad-13573790a6b5"
     dev1="9bc989d5-9003-4d56-a8b7-3e99232022d6;4"
     package="org.openintents.safe"
     dev2="899b887e-8f0a-463d-a1ad-13573790a6b5;2" ];
  5 [ label="Safe/src/org/openintents/safe/PassView.java"
     owner="9bc989d5-9003-4d56-a8b7-3e99232022d6"
```

[2]http://www.graphviz.org/doc/info/lang.html
[3]https://github.com/openintents/safe.git

```
    creator="9bc989d5-9003-4d56-a8b7-3e99232022d6"
    dev1="9bc989d5-9003-4d56-a8b7-3e99232022d6;5"
    package="org.openintents.safe"
    dev2="899b887e-8f0a-463d-a1ad-13573790a6b5;1" ];
6 [ label="SafeTest/src/org/openintents/safe/test/SafeTest.java"
    owner="dcb22e5f-3750-4aa2-9a28-4c6e2f6a4ac6"
    dev3="899b887e-8f0a-463d-a1ad-13573790a6b5;1"
    creator="9bc989d5-9003-4d56-a8b7-3e99232022d6"
    dev1="dcb22e5f-3750-4aa2-9a28-4c6e2f6a4ac6;1"
    package="org.openintents.safe.test"
    dev2="9bc989d5-9003-4d56-a8b7-3e99232022d6;1" ];
7 [ label="Safe/src/org/openintents/safe/IntentHandler.java"
    owner="9bc989d5-9003-4d56-a8b7-3e99232022d6"
    creator="9bc989d5-9003-4d56-a8b7-3e99232022d6"
    dev1="9bc989d5-9003-4d56-a8b7-3e99232022d6;3"
    package="org.openintents.safe" ];
8 [ label="Safe/src/org/openintents/safe/DBHelper.java"
    owner="9bc989d5-9003-4d56-a8b7-3e99232022d6"
    creator="9bc989d5-9003-4d56-a8b7-3e99232022d6"
    dev1="9bc989d5-9003-4d56-a8b7-3e99232022d6;4"
    package="org.openintents.safe" ];
9 [ label="Safe/src/org/openintents/safe/CategoryList.java"
    owner="9bc989d5-9003-4d56-a8b7-3e99232022d6"
    creator="899b887e-8f0a-463d-a1ad-13573790a6b5"
    dev1="9bc989d5-9003-4d56-a8b7-3e99232022d6;3"
    package="org.openintents.safe"
    dev2="899b887e-8f0a-463d-a1ad-13573790a6b5;2" ];
10 [ label="Safe/src/org/openintents/safe/Search.java"
     owner="9bc989d5-9003-4d56-a8b7-3e99232022d6"
     creator="9bc989d5-9003-4d56-a8b7-3e99232022d6"
     dev1="9bc989d5-9003-4d56-a8b7-3e99232022d6;3"
     package="org.openintents.safe" ];
11 [ label="Safe/src/org/openintents/safe/PassList.java"
     owner="9bc989d5-9003-4d56-a8b7-3e99232022d6"
     creator="9bc989d5-9003-4d56-a8b7-3e99232022d6"
     dev1="9bc989d5-9003-4d56-a8b7-3e99232022d6;2"
     package="org.openintents.safe" ];
12 [ label="Safe/src/org/openintents/safe/
        SearchListItemAdapter.java"
     owner="9bc989d5-9003-4d56-a8b7-3e99232022d6"
     creator="9bc989d5-9003-4d56-a8b7-3e99232022d6"
     dev1="9bc989d5-9003-4d56-a8b7-3e99232022d6;1"
     package="org.openintents.safe" ];
13 [ label="Safe/src/org/openintents/safe/SearchEntry.java"
```

```
      owner="9bc989d5-9003-4d56-a8b7-3e99232022d6"
      creator="9bc989d5-9003-4d56-a8b7-3e99232022d6"
      dev1="9bc989d5-9003-4d56-a8b7-3e99232022d6;1"
      package="org.openintents.safe" ];
  14 [ label="Safe/src/org/openintents/safe/Passwords.java"
      owner="9bc989d5-9003-4d56-a8b7-3e99232022d6"
      creator="9bc989d5-9003-4d56-a8b7-3e99232022d6"
      dev1="9bc989d5-9003-4d56-a8b7-3e99232022d6;2"
      package="org.openintents.safe" ];
  15 [ label="Safe/src/org/openintents/safe/CryptoHelper.java"
      owner="9bc989d5-9003-4d56-a8b7-3e99232022d6"
      creator="9bc989d5-9003-4d56-a8b7-3e99232022d6"
      dev1="9bc989d5-9003-4d56-a8b7-3e99232022d6;2"
      package="org.openintents.safe" ];
  16 [ label="Safe/src/org/openintents/safe/service/
         ServiceNotification.java"
      owner="9bc989d5-9003-4d56-a8b7-3e99232022d6"
      creator="9bc989d5-9003-4d56-a8b7-3e99232022d6"
      dev1="9bc989d5-9003-4d56-a8b7-3e99232022d6;1"
      package="org.openintents.safe.service" ];
  17 [ label="Safe/src/org/openintents/safe/SimpleGestureFilter.java"
      owner="9bc989d5-9003-4d56-a8b7-3e99232022d6"
      creator="9bc989d5-9003-4d56-a8b7-3e99232022d6"
      dev1="9bc989d5-9003-4d56-a8b7-3e99232022d6;2"
      package="org.openintents.safe" ];
  18 [ label="Safe/src/org/openintents/safe/Safe.java"
      owner="899b887e-8f0a-463d-a1ad-13573790a6b5"
      creator="899b887e-8f0a-463d-a1ad-13573790a6b5"
      dev1="899b887e-8f0a-463d-a1ad-13573790a6b5;1"
       package="org.openintents.safe" ];
  19 [ label="Safe/src/org/openintents/safe/LogOffActivity.java"
      owner="9bc989d5-9003-4d56-a8b7-3e99232022d6"
      creator="899b887e-8f0a-463d-a1ad-13573790a6b5"
      dev1="9bc989d5-9003-4d56-a8b7-3e99232022d6;1"
      package="org.openintents.safe"
      dev2="899b887e-8f0a-463d-a1ad-13573790a6b5;1" ];
  20 [ label="Safe/src/org/openintents/safe/Restore.java"
      owner="9bc989d5-9003-4d56-a8b7-3e99232022d6"
      creator="9bc989d5-9003-4d56-a8b7-3e99232022d6"
      dev1="9bc989d5-9003-4d56-a8b7-3e99232022d6;1"
      package="org.openintents.safe" ];
  21 [ label="Safe/src/org/openintents/safe/Preferences.java"
      owner="9bc989d5-9003-4d56-a8b7-3e99232022d6"
      creator="9bc989d5-9003-4d56-a8b7-3e99232022d6"
      dev1="9bc989d5-9003-4d56-a8b7-3e99232022d6;1"
      package="org.openintents.safe" ];
  22 [ label="Safe/src/org/openintents/safe/service/
         ServiceDispatchImpl.java"
      owner="9bc989d5-9003-4d56-a8b7-3e99232022d6"
      creator="9bc989d5-9003-4d56-a8b7-3e99232022d6"
      dev1="9bc989d5-9003-4d56-a8b7-3e99232022d6;1"
      package="org.openintents.safe.service" ];
  23 [ label="Safe/src/org/openintents/safe/CategoryEntry.java"
      owner="9bc989d5-9003-4d56-a8b7-3e99232022d6"
      creator="9bc989d5-9003-4d56-a8b7-3e99232022d6"
      dev1="9bc989d5-9003-4d56-a8b7-3e99232022d6;1"
      package="org.openintents.safe" ];
  24 [ label="Safe/src/org/openintents/safe/CategoryEdit.java"
      owner="9bc989d5-9003-4d56-a8b7-3e99232022d6"
      creator="9bc989d5-9003-4d56-a8b7-3e99232022d6"
      dev1="9bc989d5-9003-4d56-a8b7-3e99232022d6;1"
      package="org.openintents.safe" ];
  25 [ label="Safe/src/org/openintents/safe/Backup.java"
      owner="9bc989d5-9003-4d56-a8b7-3e99232022d6"
      creator="9bc989d5-9003-4d56-a8b7-3e99232022d6"
      dev1="9bc989d5-9003-4d56-a8b7-3e99232022d6;1"
      package="org.openintents.safe" ];
  26 [ label="Safe/src/org/openintents/safe/ChangePass.java"
      owner="9bc989d5-9003-4d56-a8b7-3e99232022d6"
      creator="9bc989d5-9003-4d56-a8b7-3e99232022d6"
      dev1="9bc989d5-9003-4d56-a8b7-3e99232022d6;1"
      package="org.openintents.safe" ];
  27 [ label="Safe/src/org/openintents/safe/
         CryptoContentProvider.java"
      owner="9bc989d5-9003-4d56-a8b7-3e99232022d6"
      creator="9bc989d5-9003-4d56-a8b7-3e99232022d6"
      dev1="9bc989d5-9003-4d56-a8b7-3e99232022d6;1"
      package="org.openintents.safe" ];
  28 [ label="Safe/src/org/openintents/safe/
         CryptoHelperException.java"
      owner="9bc989d5-9003-4d56-a8b7-3e99232022d6"
      creator="9bc989d5-9003-4d56-a8b7-3e99232022d6"
      dev1="9bc989d5-9003-4d56-a8b7-3e99232022d6;1"
      package="org.openintents.safe" ];
  1 -- 2 [ weight="1.0" ];
  7 -- 5 [ weight="1.0" ];
  9 -- 4 [ weight="2.0" ];
  9 -- 3 [ weight="1.0" ];
```

```
4 -- 3 [ weight="1.0" ];
10 -- 11 [ weight="1.0" ];
9 -- 11 [ weight="1.0" ];
9 -- 5 [ weight="2.0" ];
11 -- 5 [ weight="1.0" ];
5 -- 12 [ weight="1.0" ];
5 -- 10 [ weight="1.0" ];
5 -- 13 [ weight="1.0" ];
12 -- 10 [ weight="1.0" ];
12 -- 13 [ weight="1.0" ];
10 -- 13 [ weight="1.0" ];
4 -- 14 [ weight="1.0" ];
4 -- 8 [ weight="1.0" ];
4 -- 7 [ weight="1.0" ];
14 -- 8 [ weight="2.0" ];
14 -- 7 [ weight="2.0" ];
8 -- 7 [ weight="2.0" ];
15 -- 14 [ weight="1.0" ];
15 -- 8 [ weight="1.0" ];
15 -- 16 [ weight="1.0" ];
15 -- 7 [ weight="1.0" ];
14 -- 16 [ weight="1.0" ];
8 -- 16 [ weight="1.0" ];
16 -- 7 [ weight="1.0" ];
17 -- 5 [ weight="1.0" ];
5 -- 18 [ weight="1.0" ];
5 -- 19 [ weight="1.0" ];
18 -- 9 [ weight="1.0" ];
18 -- 19 [ weight="1.0" ];
9 -- 19 [ weight="1.0" ];
20 -- 4 [ weight="1.0" ];
20 -- 9 [ weight="1.0" ];
20 -- 21 [ weight="1.0" ];
4 -- 21 [ weight="1.0" ];
9 -- 21 [ weight="1.0" ];
22 -- 19 [ weight="1.0" ];
23 -- 24 [ weight="1.0" ];
23 -- 25 [ weight="1.0" ];
23 -- 9 [ weight="1.0" ];
24 -- 25 [ weight="1.0" ];
24 -- 9 [ weight="1.0" ];
25 -- 9 [ weight="1.0" ];
26 -- 27 [ weight="1.0" ];
26 -- 15 [ weight="1.0" ];
```

```
  26 -- 28 [ weight="1.0" ];
  27 -- 15 [ weight="1.0" ];
  27 -- 28 [ weight="1.0" ];
  15 -- 28 [ weight="1.0" ];
}
```

A.3. Commit Pattern Data

To describe the commit pattern as simple as possible, we assume that a file has exactly one class. Thus, it is enough to know what happens in detail when a file is added or changed. The two commit types *feature add* and *bugfix* are analyzed. The following data consists of average values based on the open source projects gora[4], zookeeper[5], and oisafe[6]. The mining is described in Section 5.3.1.

A.3.1. Class Changes

In Table A.1 we present the average metric changes for a new created or a updated class.

Project	Add Feature					BugFix				
	MA	MU	MD	NOI	NII	MA	MU	MD	NOI	NII
Gora	10.75	6.01	1.17	4.99	2.39	15.30	10.70	2.50	7.34	3.54
Zookeeper	6.82	4.15	1.80	4.44	2.32	0.92	2.40	0.14	3.03	0.41
Oisafe	2.66	3.12	0.77	0.90	1.29	3.05	4.00	0.74	1.00	0.27

Table A.1.: Average metric changes for a updated or created class. MA is the number of add methods, MU is the number of updated methods, MD is the number of deleted methods, NOI is the number of outgoing invocations and NII is the number of incoming invocations.

A.3.2. Method Changes

In this section the changes of common method metrics are presented for feature adds in Table A.2 and for bug fixes in Table A.3. The values represent the delta of the metrics between two commits.

[4]https://github.com/apache/gora
[5]https://github.com/apache/zookeeper
[6]https://github.com/openintents/safe

Project	LOC			McCC			NOI			NII		
	A	D	U	A	D	U	A	D	U	A	D	U
Gora	8.42	-11.74	0.13	2.02	-2.81	0.02	1.10	-1.30	0.10	0.79	-0.85	0.14
Zookeeper	11.59	11.90	0.62	2.24	-2.44	0.14	1.66	-1.52	0.16	1.16	-1.27	0.30
Oisafe	14.35	-20.02	0.02	2.67	-3.73	-0.04	0.86	-1.15	0.22	0.99	-1.17	0.04

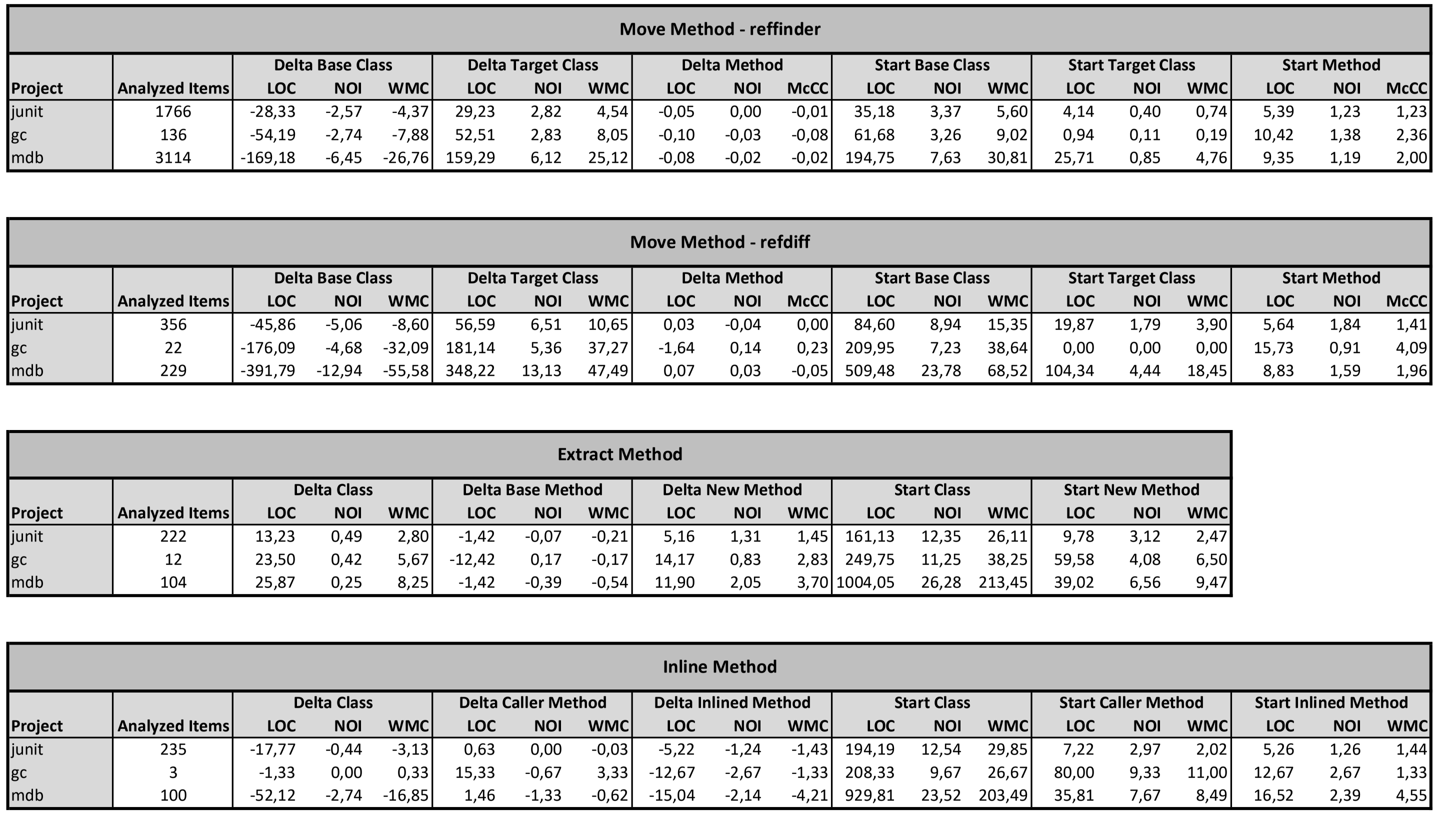
Table A.2.: Metric changes of methods during a feature add. LOC is the size measured in lines of code, McCC is the McCabe cyclomatic complexity, NOI is the number of outgoing invocations and NII is the number of incoming invocations.

Project	LOC			McCC			NOI			NII		
	A	D	U	A	D	U	A	D	U	A	D	U
Gora	8.77	-13.44	0.41	2.09	-3.02	0.13	1.19	-1.59	0.14	0.86	-1.08	0.27
Zookeeper	14.09	-12.62	1.11	2.21	-2.82	0.09	2.20	-0.87	0.13	0.93	-1.18	0.13
Oisafe	11.12	-8.50	0.68	1.95	-1.57	0.25	0.86	-1.21	0.21	0.86	-0.29	-0.01

Table A.3.: Metric changes of methods during a bug fix. LOC is the size measured in lines of code, McCC is the McCabe cyclomatic complexity, NOI is the number of outgoing invocations and NII is the number of incoming invocations.

A.4. Refactoring Data

To describe the metric changes of a applied refactoring, we analyzed the following metrics: size measured in lines of code (LOC), coupling measured in number of outgoing invocations (NOI), and complexity measured in McCabe's cyclomatic complexity (McCC for methods) and weighted methods per class (WMC for classes). The results presented in Figure A.2 are based on the mining framework presented in Section 5.3.2 and the results depicted in Figure A.2 are based on the SmartSHARK plug-in refSHARK which is described in Section 5.3.3.



Move Method - reffinder

		Delta Base Class			Delta Target Class			Delta Method			Start Base Class			Start Target Class			Start Method		
Project	Analyzed Items	LOC	NOI	WMC	LOC	NOI	WMC	LOC	NOI	McCC	LOC	NOI	WMC	LOC	NOI	WMC	LOC	NOI	McCC
junit	1766	-28,33	-2,57	-4,37	29,23	2,82	4,54	-0,05	0,00	-0,01	35,18	3,37	5,60	4,14	0,40	0,74	5,39	1,23	1,23
gc	136	-54,19	-2,74	-7,88	52,51	2,83	8,05	-0,10	-0,03	-0,08	61,68	3,26	9,02	0,94	0,11	0,19	10,42	1,38	2,36
mdb	3114	-169,18	-6,45	-26,76	159,29	6,12	25,12	-0,08	-0,02	-0,02	194,75	7,63	30,81	25,71	0,85	4,76	9,35	1,19	2,00

Move Method - refdiff

		Delta Base Class			Delta Target Class			Delta Method			Start Base Class			Start Target Class			Start Method		
Project	Analyzed Items	LOC	NOI	WMC	LOC	NOI	WMC	LOC	NOI	McCC	LOC	NOI	WMC	LOC	NOI	WMC	LOC	NOI	McCC
junit	356	-45,86	-5,06	-8,60	56,59	6,51	10,65	0,03	-0,04	0,00	84,60	8,94	15,35	19,87	1,79	3,90	5,64	1,84	1,41
gc	22	-176,09	-4,68	-32,09	181,14	5,36	37,27	-1,64	0,14	0,23	209,95	7,23	38,64	0,00	0,00	0,00	15,73	0,91	4,09
mdb	229	-391,79	-12,94	-55,58	348,22	13,13	47,49	0,07	0,03	-0,05	509,48	23,78	68,52	104,34	4,44	18,45	8,83	1,59	1,96

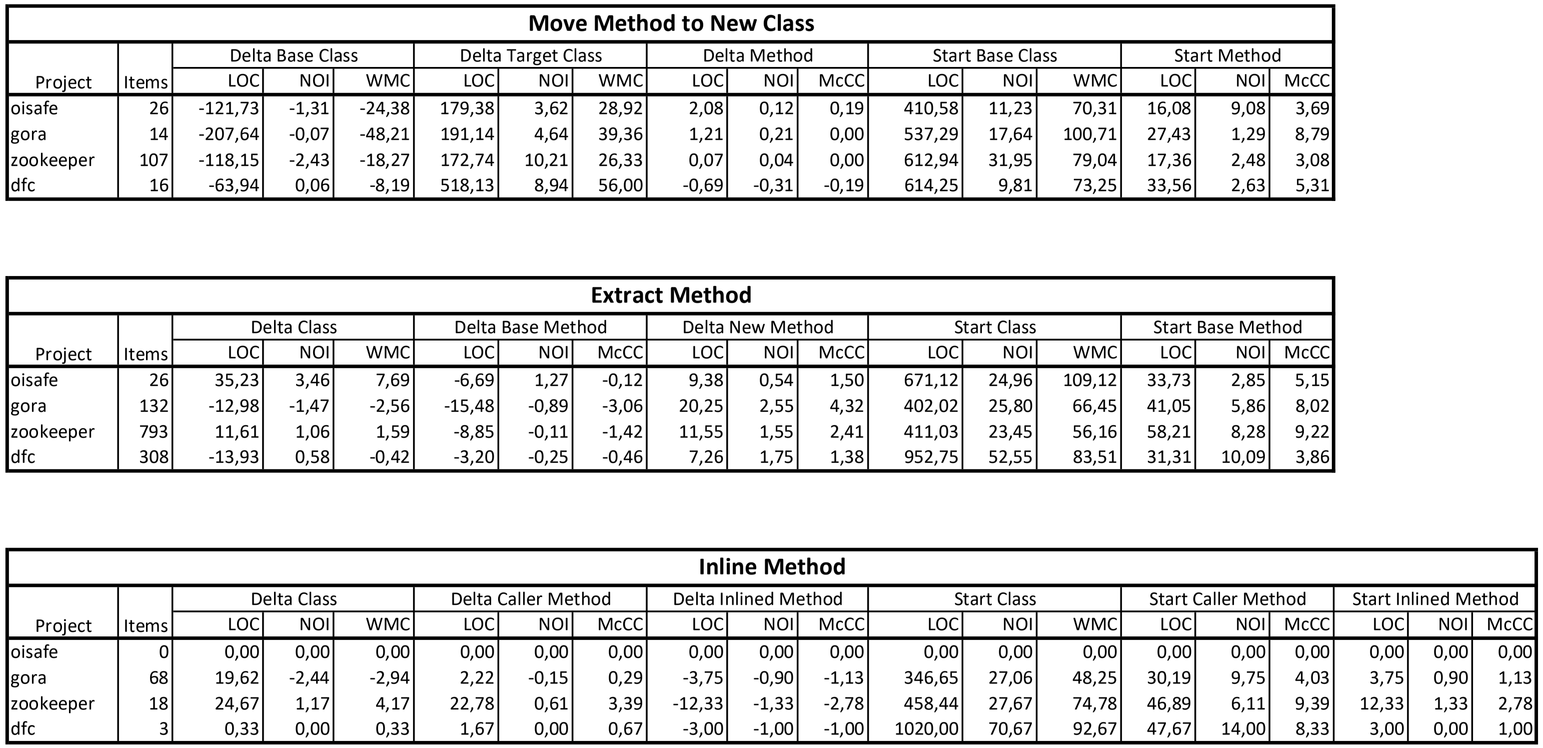
Extract Method

		Delta Class			Delta Base Method			Delta New Method			Start Class			Start New Method		
Project	Analyzed Items	LOC	NOI	WMC	LOC	NOI	WMC	LOC	NOI	WMC	LOC	NOI	WMC	LOC	NOI	WMC
junit	222	13,23	0,49	2,80	-1,42	-0,07	-0,21	5,16	1,31	1,45	161,13	12,35	26,11	9,78	3,12	2,47
gc	12	23,50	0,42	5,67	-12,42	0,17	-0,17	14,17	0,83	2,83	249,75	11,25	38,25	59,58	4,08	6,50
mdb	104	25,87	0,25	8,25	-1,42	-0,39	-0,54	11,90	2,05	3,70	1004,05	26,28	213,45	39,02	6,56	9,47

Inline Method

		Delta Class			Delta Caller Method			Delta Inlined Method			Start Class			Start Caller Method			Start Inlined Method		
Project	Analyzed Items	LOC	NOI	WMC	LOC	NOI	WMC	LOC	NOI	WMC	LOC	NOI	WMC	LOC	NOI	WMC	LOC	NOI	WMC
junit	235	-17,77	-0,44	-3,13	0,63	0,00	-0,03	-5,22	-1,24	-1,43	194,19	12,54	29,85	7,22	2,97	2,02	5,26	1,26	1,44
gc	3	-1,33	0,00	0,33	15,33	-0,67	3,33	-12,67	-2,67	-1,33	208,33	9,67	26,67	80,00	9,33	11,00	12,67	2,67	1,33
mdb	100	-52,12	-2,74	-16,85	1,46	-1,33	-0,62	-15,04	-2,14	-4,21	929,81	23,52	203,49	35,81	7,67	8,49	16,52	2,39	4,55

Figure A.1.: Mining results based on the old framework. Average metric changes of analyzed refactoring types.



Move Method to New Class

Project	Items	Delta Base Class			Delta Target Class			Delta Method			Start Base Class			Start Method		
		LOC	NOI	WMC	LOC	NOI	WMC	LOC	NOI	McCC	LOC	NOI	WMC	LOC	NOI	McCC
oisafe	26	-121,73	-1,31	-24,38	179,38	3,62	28,92	2,08	0,12	0,19	410,58	11,23	70,31	16,08	9,08	3,69
gora	14	-207,64	-0,07	-48,21	191,14	4,64	39,36	1,21	0,21	0,00	537,29	17,64	100,71	27,43	1,29	8,79
zookeeper	107	-118,15	-2,43	-18,27	172,74	10,21	26,33	0,07	0,04	0,00	612,94	31,95	79,04	17,36	2,48	3,08
dfc	16	-63,94	0,06	-8,19	518,13	8,94	56,00	-0,69	-0,31	-0,19	614,25	9,81	73,25	33,56	2,63	5,31

Extract Method

Project	Items	Delta Class			Delta Base Method			Delta New Method			Start Class			Start Base Method		
		LOC	NOI	WMC	LOC	NOI	McCC	LOC	NOI	McCC	LOC	NOI	WMC	LOC	NOI	McCC
oisafe	26	35,23	3,46	7,69	-6,69	1,27	-0,12	9,38	0,54	1,50	671,12	24,96	109,12	33,73	2,85	5,15
gora	132	-12,98	-1,47	-2,56	-15,48	-0,89	-3,06	20,25	2,55	4,32	402,02	25,80	66,45	41,05	5,86	8,02
zookeeper	793	11,61	1,06	1,59	-8,85	-0,11	-1,42	11,55	1,55	2,41	411,03	23,45	56,16	58,21	8,28	9,22
dfc	308	-13,93	0,58	-0,42	-3,20	-0,25	-0,46	7,26	1,75	1,38	952,75	52,55	83,51	31,31	10,09	3,86

Inline Method

Project	Items	Delta Class			Delta Caller Method			Delta Inlined Method			Start Class			Start Caller Method			Start Inlined Method		
		LOC	NOI	WMC	LOC	NOI	McCC	LOC	NOI	McCC	LOC	NOI	WMC	LOC	NOI	McCC	LOC	NOI	McCC
oisafe	0	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00
gora	68	19,62	-2,44	-2,94	2,22	-0,15	0,29	-3,75	-0,90	-1,13	346,65	27,06	48,25	30,19	9,75	4,03	3,75	0,90	1,13
zookeeper	18	24,67	1,17	4,17	22,78	0,61	3,39	-12,33	-1,33	-2,78	458,44	27,67	74,78	46,89	6,11	9,39	12,33	1,33	2,78
dfc	3	0,33	0,00	0,33	1,67	0,00	0,67	-3,00	-1,00	-1,00	1020,00	70,67	92,67	47,67	14,00	8,33	3,00	0,00	1,00

Figure A.2.: Mining results based on refSHARK. Average metric changes of analyzed refactoring types.

B. Simulation at Runtime

Contents

This appendix describes parameters that can be adjusted in the simulation environment before each simulation run. Furthermore, some example views of the running simulation provided by Repast Simphony are depicted.

B.1. Simulation Parameters at Runtime

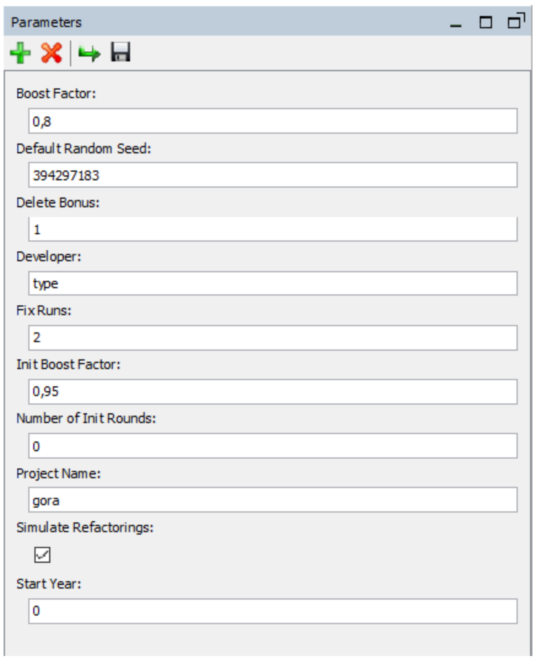
Repast Symphony allows us to define parameters that can be changed in the running application before each simulation run. For the models presented in this thesis, we have defined parameters depicted in Figure B.1.

In the following we describe each parameter in more detail. Here, the parameters are ordered according to their importance and not alphabetically as in the simulation.

Project Name The name of the project to simulate. According to the name, the configuration files with mined parameters are read.

Start Year The starting point of the simulation. If the year is set to 0, then the simulation starts with an empty change coupling graph. Otherwise, the simulation is initialized with the change coupling graph of the given year. For this the mined graph is loaded.

Developer This parameter represents the developer type to use for the next simulation run. If developer is set to *type*, then developers are divided into key, maintainer, major, and minor. If this parameter is set to *role*, then developers are classified into core and peripheral according to the onion model.



Boost Factor:
0,8
Default Random Seed:
394297183
Delete Bonus:
1
Developer:
type
Fix Runs:
2
Init Boost Factor:
0,95
Number of Init Rounds:
0
Project Name:
gora
Simulate Refactorings:
☑
Start Year:
0

Figure B.1.: Adjustable simulation parameters.

Number of Init Rounds If the mining reveals two different update behaviors, for example, an initial phase with pronounced growth and, afterwards, a phase with significantly less growth, then this parameter represents the duration of the initial phase.

Initial Boost Factor This parameter adjusts the effort that the developers spend in the initial phase.

Boost Factor This parameter adjusts the effort that the developers spend in the second phase.

Delete Bonus If not enough files are deleted by the developers work, deleting files can be rewarded using this parameter.

Fix Runs Due to the fact that the bug creation probabilities are based on ITSs data and the bugfix probabilities of the developers are based on VCSs discrepancies between the number of created and the number of fixed bugs can occur. This parameter adjusts the number of trials a developer tries to fix a bug.

Simulate Refactorings If this parameter is switched on, software refactorings are additionally simulated. This can slow down the runtime.

Default Random Seed Default parameter of Repast Simphony used for random number generation.

B.2. Simulation Views at Runtime

In this section, we present two views of the simulation environment at runtime.

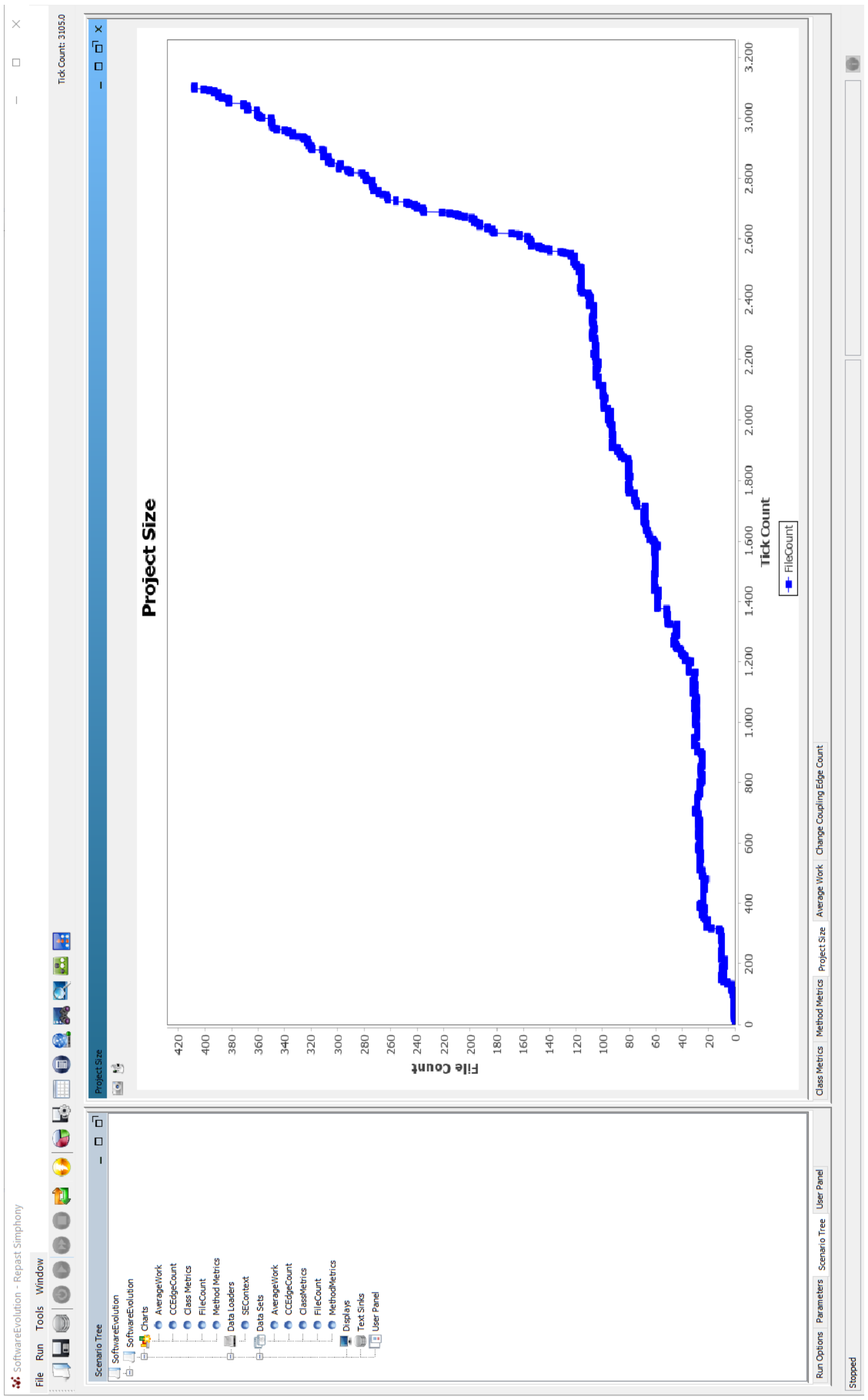
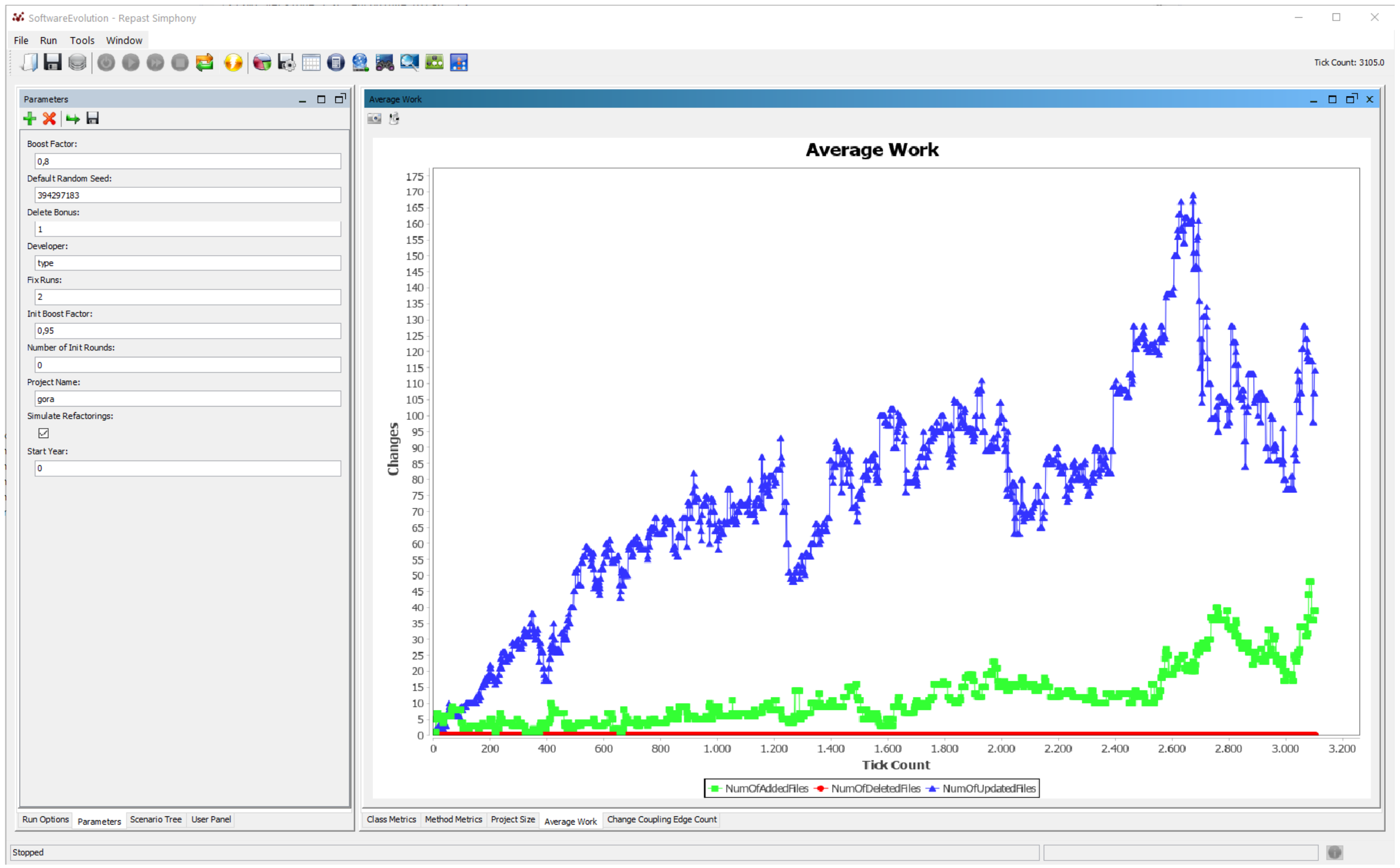


Figure B.2.: Simulation at runtime with the scenario tree on the left side and the live view of the project growth on the right side.



SoftwareEvolution - Repast Simphony

Parameters

Boost Factor: 0,8

Default Random Seed: 394297183

Delete Bonus: 1

Developer: type

Fix Runs: 2

Init Boost Factor: 0,95

Number of Init Rounds: 0

Project Name: gora

Simulate Refactorings: ☑

Start Year: 0

Tick Count: 3105.0

Stopped

Figure B.3.: Simulation at runtime with the adjustable parameters on the left side and the live view of the average work of all developers by round on the right side.

C. Mining Implementation Details

Contents

Here, we present details of our mining implementations. First, we introduce some basics about the used mapping tool Morphia [143] in Section C.1. Afterwards, we present the used data model of our mining applications based on SmartSHARK in Section C.1.

C.1. MongoDB with Morphia – A Basic Introduction

Since the SmartSHARK database is a MongoDB [144], we use Morphia as JVM object document mapper. By using Morphia, we can easily use all the classes shown here like POJOs in the code.

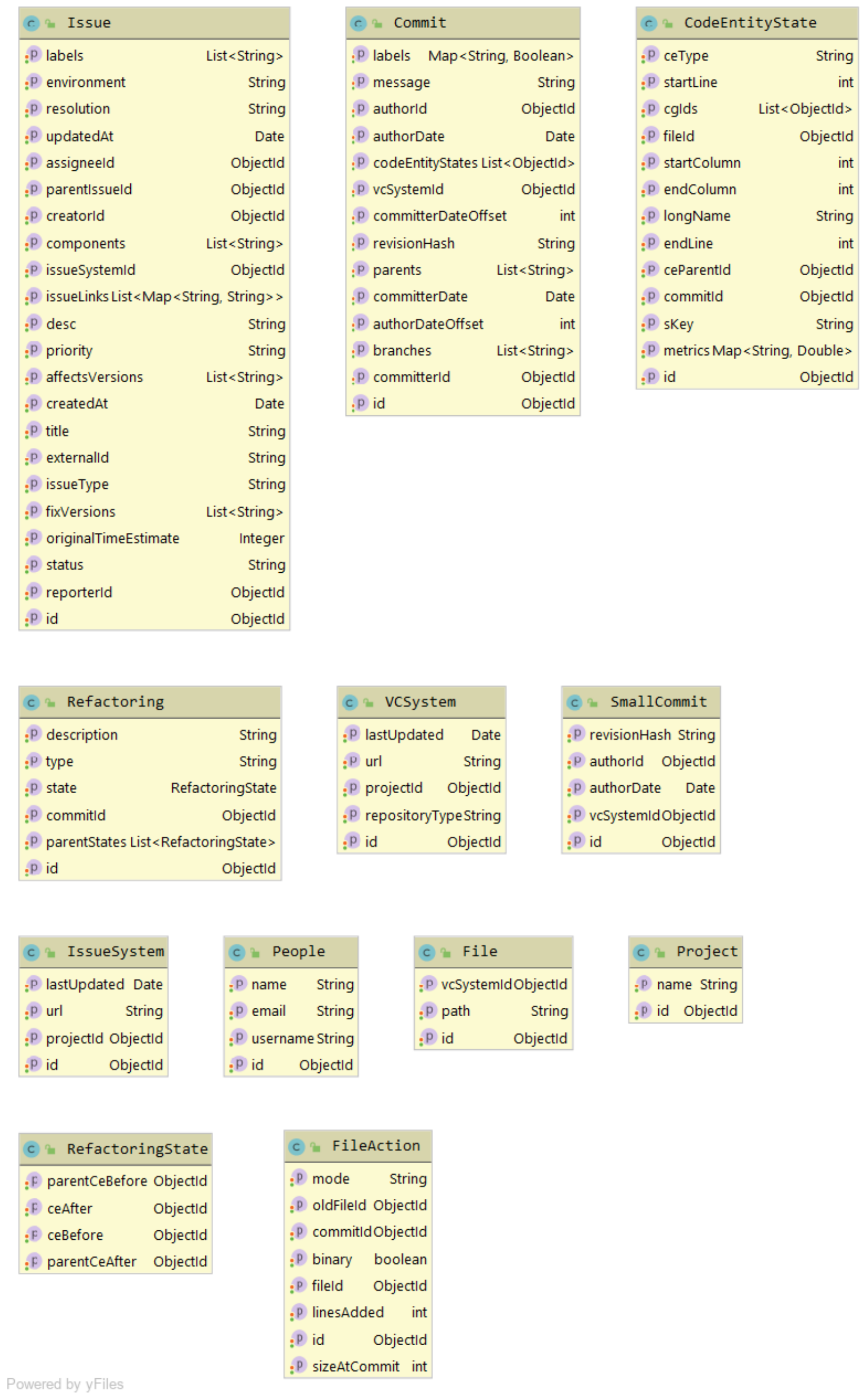
To connect to a MongoDB with Morphia mainly two things are required. Firstly, a *Morphia* instance must be created. This class configures among other things the mapper. Secondly, the *Datastore* is created using the Morphia instance. For this, a suitable connection string is required as described in the documentation.

Afterwards, we can save and query data using the datastore instance. This allows us to iterate easily over, for example, all commits of a project.

C.2. Used classes of the SmartSHARK data model

In this section, we present the data model used for all our mining implementations based on SmartSHARK.

The model presented in Figure C.1 contains all classes we use for our mining work based on SmartSHARK in this thesis.



Issue	
labels	List<String>
environment	String
resolution	String
updatedAt	Date
assigneeId	ObjectId
parentIssueId	ObjectId
creatorId	ObjectId
components	List<String>
issueSystemId	ObjectId
issueLinks	List<Map<String, String>>
desc	String
priority	String
affectsVersions	List<String>
createdAt	Date
title	String
externalId	String
issueType	String
fixVersions	List<String>
originalTimeEstimate	Integer
status	String
reporterId	ObjectId
id	ObjectId

Commit	
labels	Map<String, Boolean>
message	String
authorId	ObjectId
authorDate	Date
codeEntityStates	List<ObjectId>
vcSystemId	ObjectId
committerDateOffset	int
revisionHash	String
parents	List<String>
committerDate	Date
authorDateOffset	int
branches	List<String>
committerId	ObjectId
id	ObjectId

CodeEntityState	
ceType	String
startLine	int
cgIds	List<ObjectId>
fileId	ObjectId
startColumn	int
endColumn	int
longName	String
endLine	int
ceParentId	ObjectId
commitId	ObjectId
sKey	String
metrics	Map<String, Double>
id	ObjectId

Refactoring	
description	String
type	String
state	RefactoringState
commitId	ObjectId
parentStates	List<RefactoringState>
id	ObjectId

VCSystem	
lastUpdated	Date
url	String
projectId	ObjectId
repositoryType	String
id	ObjectId

SmallCommit	
revisionHash	String
authorId	ObjectId
authorDate	Date
vcSystemId	ObjectId
id	ObjectId

IssueSystem	
lastUpdated	Date
url	String
projectId	ObjectId
id	ObjectId

People	
name	String
email	String
username	String
id	ObjectId

File	
vcSystemId	ObjectId
path	String
id	ObjectId

Project	
name	String
id	ObjectId

RefactoringState	
parentCeBefore	ObjectId
ceAfter	ObjectId
ceBefore	ObjectId
parentCeAfter	ObjectId

FileAction	
mode	String
oldFileId	ObjectId
commitId	ObjectId
binary	boolean
fileId	ObjectId
linesAdded	int
id	ObjectId
sizeAtCommit	int

Figure C.1.: Data model used for tools based on SmartSHARK.