Developer-Centric Software Assessment

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

Philip Makedonski
aus Smolyan, Bulgarien

Göttingen, 2018

Abstract

Software systems are becoming more and more widespread in all areas of everyday life. Due to the increasing reliance on such systems, there is a need to keep them operational over longer periods of time under constantly changing circumstances and increasing demands. Thus, it becomes essential to develop and maintain software with an evolutionary mindset. Various kinds of software assessment are employed to gain a better understanding of the nature of software evolution and provide methods and tools to support the evolution of software. *Artifact-centric* assessment captures the state of affairs at a given point in time as reflected in the characteristics of the different artifacts that comprise a software system. *Change-centric* assessment, in contrast, considers how a software system evolved into the state it is at a given point in time and how it can be expected to evolve in the future. Since changes do not occur by themselves, in this thesis we shift to focus to the developers performing the changes, by proposing *developer-centric* software assessment.

The overarching goal of this thesis is to investigate means for characterising developer contribution behaviour and assessing its impact on the resulting software products with respect to certain events of interest. The characterisation and assessment are based on traces collected from different kinds of software-related assets, containing information related to software artifacts at different levels of granularity. Pursuing this goal, we make several contributions within the scope of this thesis, which are related to the identification of potential causes for events of interest and the characterisation of developer behaviour, as well as a model-based approach for mining software repositories and conducting software assessment. We perform case studies to evaluate the methods described in the thesis.

The approach for the identification of potential causes for events of interest adds quantitative information on top of existing approaches for origin analysis in order to provide more accurate information across multiple levels of granularity. The approach for the characterisation of developer behaviour seeks to capture and assess the circumstances in which development activities are performed. We present a selection of characteristics across different dimensions and discussed different approaches for making use of the resulting data based on visualisation and data mining techniques. Both approaches are realised within a model-based software mining infrastructure aiming to ease the integration of heterogeneous data produced and used by third-party tools. It serves as a glue for loosely coupling software mining solutions at a high level of abstraction. The corresponding case studies demonstrate the application of the approaches and their strengths and limitations.

Acknowledgements

During the course of my PhD studies, I have been very fortunate to benefit from the influence of many people in various ways. All of them have played an important role in shaping my work, in my personal development and growth, and in my life as a whole. I would like to take this opportunity to acknowledge their contributions and thank them for their support throughout the last couple of years. People are listed in no particular order.

First, I would like to thank my mentors and advisors, Prof. Dr. Jens Grabowski and Prof. Dr. Stephan Waack, for their continued support, insightful comments and valuable suggestions. Their knowledge and ideas contributed a lot to the results of this thesis and to my personal growth throughout the years. They provided numerous opportunities for personal and professional development and I am very grateful for all the experience and knowledge I gathered as a result.

My colleagues in Göttingen always provided an enjoyable, supporting, and scientifically inspiring work environment: Annette Kadziora, Heike Jachinke, Gunnar Krull, Udo Burghardt, Steffen Herbold, Fabian Trautsch, Alexander Trautsch, Johannes Martin Erbel, Fabian Gumz, Ella Albrecht, Verena Herbold, Patrick Harms, Fabian Korte, Benjamin Zeiss, Thomas Rings, Edith Werner, Wafi Dahman, Akthar Ali Jalbani, Xiaowei Wang, Fabian Sudau, Helmut Neukirchen, Daniel Honsel. I am especially grateful for the efforts that Steffen Herbold, Fabian Trautsch and Fabian Korte put into the proof reading of this thesis. I would also like to thank all of my collaborators from the various projects and initiatives that I was involved in, they provided numerous insights and learning opportunities.

Many many thanks to my family that has always supported me unconditionally during my studies. Also big thanks to my friends for their support and for providing new perspectives on many aspects related to the work on this thesis and beyond.

Contents

List of Figures

List of Tables

Listings

Acronyms

ABM *Agent-Based Modelling*

AMDD *Agile Model Driven Development*

API *Application Programming Interface*

ARFF *Attribute-Relation File Format*

AST *Abstract Syntax Tree*

CI *Continuous Integration*

CLI *Command Line Interface*

CSV *Comma Separated Values*

DAG *Directed Acyclic Graph*

DSL *Domain-Specific Language*

DSM *Domain-Specific Modeling*

EBNF *Extended Backus–Naur Form*

EMF *Eclipse Modeling Framework*

EMOF *Essential Meta-Object Facility*

EOL *Epsilon Object Language*

ETL *Epsilon Transformation Language*

ETSI *European Telecommunications Standards Institute*

FAMIX *FAMOOS Information Exchange Model*

GMF *Graphical Modeling Framework*

GQM *Goal Question Metric*

GUI *Graphical User Interface*

HMM	*Hidden Markov Model*
IDE	*Integrated Development Environment*
IDE	*Integrated Development Environment*
ITS	*Issue Tracking System*
KDE	*K Desktop Environment*
LOC	*Lines of Code*
M2M	*Model-to-Model*
M2T	*Model-to-Text*
MBE	*Model Based Engineering*
MDA	*Model Driven Architecture*
MDD	*Model Driven Development*
MDE	*Model Driven Engineering*
MLA	*Mailing List Archive*
MOF	*Meta-Object Facility*
MSR	*Mining Software Repositories*
OCL	*Object Constraint Language*
OMG	*Object Management Group*
ORM	*Object Relational Mapping*
OWL	*Web Ontology Language*
QVT	*Query / View / Transformation*
SWT	*Standard Widget Toolkit*
TDL	*Test Description Language*
TTCN-3	*Testing and Test Control Notation*
UML	*Unified Modelling Language*
URI	*Uniform Resource Identifier*

VCS	*Version Control System*
XMI	*XML Metadata Interchange*
XML	*Extensible Markup Language*

1. Introduction

Software systems are becoming more and more widespread in all areas of everyday life. With the increasing reliance on software for accomplishing various tasks as well as keeping equipment operational for longer and often unforeseen length of time, it becomes necessary to maintain and evolve software for long periods of time, often well beyond the originally anticipated lifespan of the software. In addition to the longer lifespan of software, the requirements for the software also change over time and with the increasingly widespread adoption, they change more frequently due to several related factors. The broader user-base has more and diverse usage scenarios requiring new and extended functionality. The broader user-base is also more sensitive to shortcomings and defects in the software which have a larger impact and potential for disruption. As a consequence, software systems are gaining more and more attention from regulatory bodies which push further requirements on the development and operation of software. The diversified usage scenarios, the pressure from users and legislation, along with the rapidly changing hardware and software landscape in which a software system operates and depends on lead to continuously changing requirements. The realisation of the changing requirements leads to changes in the artifacts related to the software, which also manifest in changes in the characteristics of the software artifacts.

The field of *software evolution* is concerned with studying and understanding the continuous change of software systems over time. This is achieved by collecting observations on how things changed in the past, predicting how things are likely to change in the future based on the observations from the past, and guiding decisions about the development and maintenance of the software the present based on the past observations and the future predictions. Thus, on the one hand, software evolution is concerned with understanding the nature of the evolution phenomenon and its underlying drivers. On the other hand, software evolution is concerned with the achievement of evolution by providing methods, tools, and techniques for changing characteristics of the software in a controlled, disciplined, reliable, fast, and cost-effective manner [107]. In order to gain a better understanding of the nature of software evolution and provide methods and tools for the achievement of evolution, various kinds of software assessment are employed.

Software assessment is the process of posing specific questions about the software system under study and carrying out specialised analyses to answer these questions [131]. When assessing software systems, the most intuitive approach is to look at the software itself and contemplate the characteristics of the different artifacts that comprise the software. This approach, which we will refer to as *artifact-centric* assessment, captures the state of affairs at a given point in time. As software systems are continuously evolving, it becomes more and more important to ask questions related to how the software evolved into the state it is at a given point in time and how it can be expected to evolve in the future. When considering such questions, we are also concerned with how the software changed over time rather than only how it is at the certain point in time. In this case we are speaking of *change-centric* assessment. But changes do not occur by themselves. Changes are performed by developers.

Similar to the use of software, the development of software is also becoming more and more widespread and diversified. With the rise of open source software, the very nature of software development has changed fundamentally. People from all walks of life can contribute to software development, regardless of their training, experience, and location. They may choose to contribute only towards a specific requirement that they are affected by, dedicate continuous contributions to a project, or even be required to contribute by organisational policies. Depending on the individual strengths and weaknesses of each developer as well as on the collaborations between different developers, the contributions may have a different impact on the software. The amount, scope, and impact of contributions, as well as the reasons and motivations for the contributions may also change over time. Finally, the experience and working habits of the developers are also likely to change over time. All of this can have an effect on the governance of the software project and the organisation and coordination of the contributions, resulting in different guidelines, policies, roles, and processes for contributors. In order to better understand how developer contribution behaviour evolves over time as a core factor in determining how software evolves over time, we propose *developer-centric* software assessment. After looking at artifacts and at changes to the artifacts over time, it is now time to look at the developers behind the changes. Based on developer-centric software assessment, tools can provide relevant feedback specific to a particular developer under particular circumstances. Developer-centric software assessment can also yield potentially helpful insights for guiding organisational decisions.

Existing research has shown some promise and potential benefits of considering developer-related information and building developer-specific models for identifying risky changes [163] and personalised defect prediction [84]. This thesis pursues this direction further bringing developers and developer-related information even more to the forefront of software assessment.

1.1. Goals and Contributions

The overarching goal of this thesis is to investigate means for characterising developer contribution behaviour and its evolution, as well as assessing its impact on different aspects of the evolution of software. The characterisation and assessment is based on traces collected from different kinds of software-related assets, such as *Version Control System*s (VCSs) and *Issue Tracking System*s (ITSs).

The work towards this goal is guided by the following high-level research questions and addressing subsequent challenges associated with these questions:

- How can we characterise developer behaviour based on information collected from software-related assets?
- How can we determine potential causes for events of interest across multiple levels of abstraction?
- How can we mine information related to developer behaviour and its impact on potential causes for events of interest from software-related assets in an effective and agile manner?
- What are the advantages of using developer-centric software assessment?

The first challenge with regard to characterising developer behaviour is to *determine what constitutes developer behaviour*. We need to consider the different circumstances under which developers operate. This includes various sources of information, different levels of granularity, as well as collaborations with other developers. The circumstances are characterised by both situational factors related to the artifacts on which a developer works as well as dispositional factors related to the developer working on the artifacts.

The next challenge is to *assess the impact of the contribution behaviour* with respect to certain events of interest and their potential causes. Events of interest could be bug fixes or refactorings, for example. In order to better understand the potential causes for these events of interest, such as introducing defects and smells, we contemplate the circumstances that are associated with them.

The third challenge is to *investigate the impact of changes in the behaviour of developers*. Developers gain experience over time. They may also become more or less involved in a project over time. Consequently, different developers may assume different roles over time or also express different modes of operation while assuming the same role. This may affect the outcome of their activities with respect to potential causes for events of interest.

The fourth challenge is to *investigate transfer opportunities between different developers and different projects*, e.g., in new projects for which there is no sufficient data available for assessment, or in existing projects when new developers join the project.

Different assessment applications, such as defect prediction, risk assessment, software process simulation, and visualisation can benefit from better characterisation of the developers over time. Such applications can be utilised to guide organisational decisions and also provide relevant feedback to developers that is specific to the circumstances at a particular point in time.

In order to tackle the research questions and challenges noted above, the contributions of this thesis can be summarised as follows:

- A method for characterising developer behaviour from both situational and dispositional perspectives, considering different facets, sources of information, levels of granularity, as well as potential changes in the behaviour of developers.
- A method for identifying potential causes for events of interest across different levels of granularity.
- A model-based approach for mining software repositories and conducting software assessment with a concrete instantiation for developer-centric software assessment.
- Case studies for the evaluation of the methods described in this thesis, including refinements of the research questions to target specific aspects.

1.2. Impact

The results of this thesis as well as work related to the application of the methods discussed in this thesis have been published in several peer-reviewed journals, international workshop and conference proceedings, as well as book chapters, including:

Journal Articles

- Fabian Trautsch, Steffen Herbold, Philip Makedonski, Jens Grabowski *Addressing problems with replicability and validity of repository mining studies through a smart data platform*, Empirical Software Engineering, Springer, 2017
- Philip Makedonski, Jens Grabowski *Testbeschreibung mit TDL: Konzepte und Notationen der ETSI Test Description Language*, OBJEKTspektrum Online Themenspezial: Testing, SIGS DATACOM, 2016
- Janka Koschack, Lara Weibezahl, Tim Friede, Wolfgang Himmel, Philip Makedonski, Jens Grabowski. *Scientific versus experiential evidence: Discourse analysis of the CCSVI debate in a multiple sclerosis forum*, Journal of Medical Internet Research, JMIR - Publications, http://www.jmir.org/2015/7/e159/, 2015
- Philip Makedonski, Fabian Sudau, Jens Grabowski. *Towards a Model-based Software Mining Infrastructure*, ACM SIGSOFT Software Engineering Notes 40(1), ACM, 2015
- Fabian Sudau, Tim Friede, Jens Grabowski, Janka Koschack, Philip Makedonski, Wolfgang Himmel. *Sources of Information and Behavioral Patterns in Health Online Forums*, Journal of Medical Internet Research, JMIR - Publications, http://www.jmir.org/2014/1/e10/, 2014
- Philip Makedonski, Jens Grabowski, Florian Philipp. *Quantifying the evolution of TTCN-3 as a language*, International Journal on Software Tools for Technology Transfer (STTT). (ISSN 1433-2779) DOI: 10.1007/s10009-013-0282-1, Springer-Verlag Berlin Heidelberg, 2013

- Jens Grabowski, Philip Makedonski, Thomas Rings, Benjamin Zeiß. *Systematische Qualitätssicherung für Testartefakte*, OBJEKTspektrum Online Themenspezial: Testing, SIGS DATACOM, 2009

Articles in Conference Proceedings

- Philip Makedonski, Gusztav Adamis, Martti Käärik, Finn Kristoffersen, Xavier Zeitoun. *Evolving the ETSI Test Description Language*, Proceedings of the 9th System Analysis and Modelling Conference (SAM 2016), Springer, 2016
- Fabian Trautsch, Steffen Herbold, Philip Makedonski, Jens Grabowski. *Addressing Problems with External Validity of Repository Mining Studies Through a Smart Data Platform*, 13th International Conference on Mining Software Repositories, 2016
- Philip Makedonski, Jens Grabowski. *Weighted Multi-Factor Multi-Layer Identification of Potential Causes for Events of Interest in Software Repositories*, To appear in: Proceedings of the Seminar Series on Advanced Techniques & Tools for Software Evolution (SATToSE), 2015
- Philip Makedonski, Helmut Neukirchen, Jens Grabowski. *Validating the Behavioral Equivalence of TTCN-3 Test Cases*, First International Conference on Advances in System Testing and Validation Lifecycle (VALID 2009), IEEE, 2009

Book Chapters

- Philip Makedonski, Tim Friede, Jens Grabowski, Janka Koschack, Wolfgang Himmel. *Sources of Information and Behavioural Patterns in Health Online Fora*, To appear in: Social Network Analysis: Interdisciplinary Approaches and Case Studies, CRC Press, 2016
- 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*, To appear in: Social Network Analysis: Interdisciplinary Approaches and Case Studies, CRC Press, 2016
- Jürgen Großmann, Philip Makedonski, Hans-Werner Wiesbrock, Jaroslav Svacina, Ina Schieferdecker, Jens Grabowski. *Model-Based X-in-the-Loop Testing*, Model-Based Testing for Embedded Systems (Computational Analysis, Synthesis, and Design of Dynamic Systems Series), CRC Press, 2011

In addition, during the work on this thesis, the author defined and supervised three master's theses, one bachelor's thesis, and several students' projects on topics related to this thesis, including:

Master's Theses

- Sonja Neue. *Determining Test Focus and Priorities in an Industrial Environment*, Masterarbeit im Studiengang Angewandte Informatik am Institut für Informatik, ZAI-MSC-2014-04, 1612-6793, Zentrum für Informatik, Georg-August-Universität Göttingen, 2014
- Florian Philipp. *Model-diven Language Implementation Using the Example of a Test Description Language*, Masterarbeit im Studiengang Angewandte Informatik am Institut für Informatik, ZFI-MSC-2013-04, ISSN 1612-6793, Zentrum für Informatik, Georg-August-Universität Göttingen, 2013
- Fabian Sudau. *Analysis of Controversial Debates in Online Fora - A Showcase Analysis of the CCSVI Discussion in the DMSG Layperson Forum*, Masterarbeit im Studiengang Angewandte Informatik am Institut für Informatik, ZAI-MSC-2013-04, ISSN 1612-6793, Zentrum für Informatik, Georg-August-Universität Göttingen, 2013

Bachelor's Theses

- Daniel May. *Observing Activity Patterns in Software Development*, Bachelorarbeit im Studiengang Angewandte Informatik am Institut für Informatik, ZAI-BSC-2012-07, ISSN 1612-6793, Zentrum für Informatik, Georg-August-Universität Göttingen, 2012

The author has also actively contributed to the standardisation of the *Test Description Language* (TDL) at *European Telecommunications Standards Institute* (ETSI), co-authoring all sixteen versions of the seven standards related to the language. The tools and technologies developed during the work on this thesis have been integrated into further projects, such as SmartSHARK[1], the TDL open source project[2], the *Testing and Test Control Notation* (TTCN-3) guideline checking and documentation generation tools[3], and a platform for assuring software quality by means of simulation methods[4], fostering further work and continued development of the methods and tools described in this thesis.

1.3. Thesis Structure

This thesis covers several aspects related to the central topic of developer-centric software assessment. It is structured as follows:

[1]See `https://smartshark2.informatik.uni-goettingen.de`

[2]See `https://tdl.etsi.org/index.php/open-source`

[3]See `https://t3tools.informatik.uni-goettingen.de/trac`

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

Chapter 2 summarises the essential background information that is necessary for understanding the rest of this thesis, including software evolution (Section 2.1), data mining and data analytics (Section 2.2), mining software repositories (Section 2.3), and modelling and model driven engineering (Section 2.4).

Chapter 3 is concerned with identifying likely causes for events of interest based on information derived by applying a line-tracking approach (Section 3.1). The approach relies on a generic framework (Section 3.2) to qualify events as *fixes* and determine their likely *causes*, where weights are calculated for different factors (Section 3.3) and distributed across different levels of granularity (Section 3.4) according to various weight distribution strategies (Section 3.5) emphasising different characteristics of the states.

Chapter 4 contains the description of the behaviour characterisation methodology, including a conceptual overview (Section 4.1), a description of the different characteristics across the various dimensions (Section 4.2), and means for making sense of the collected data and gaining further insights (Section 4.3).

Chapter 5 describes a model-based approach to software mining, outlining mining challenges (Section 5.1), building a case for a model-based mining approach (Section 5.2), detailing the mining process and overall framework of the approach in general terms (Section 5.3), and presenting a concrete instantiation of the mining approach for the purposes of developer-centric software assessment (Section 5.4).

Chapter 6 presents the case studies used for the evaluation of the methods described in this thesis, including the specific goals for the evaluation (Section 6.1), the evaluation criteria (Section 6.2), a description of the data sets (Section 6.3), and the case study results (Section 6.4).

Chapter 7 includes a discussion of the results from the case studies and their interpretation (Section 7.1), a comparison to related approaches (Section 7.2), assessment of the strengths and limitations of the different methods described in this thesis and their realisation (Section 7.3), as well as potential threats to validity (Section 7.4).

Chapter 8 concludes this thesis with a summary and an outlook on future work.

2. Background

In this chapter, we describe the background information related to this thesis and the broader context of the work. We first present a brief introduction to the field of software evolution as the main domain for the present work. Then, we include short summaries of relevant aspects from related domains, including data mining and data analytics, mining software repositories, as well as modelling and model based engineering, which form the foundations of the approaches presented in this thesis.

2.1. Software Evolution

Software evolution [104, 105, 106] describes the phenomenon of continuous change to software systems in order to maintain their usefulness and operability in continuously changing environments. With changing operational environments, including changes to hardware, related software, legislation, business and user needs, the requirements of the software system change as well to address the changes in the environment. The artifacts that make up the software system also need to be changed as a consequence. The different artifacts are characterised by a number of properties which in turn change as well, as the artifacts themselves are subjected to changes corresponding to the changes in the requirements. A better understanding of the nature of these changes in the artifacts and in their characteristics can be inferred based on observations of how the software evolved the past. This understanding can then be used in the assessment of certain characteristics of interest projected into the future. Characteristics of interest typically include estimated growth and associated effort as well as resulting complexity and potential risks for defects or other undesirable consequences. Insights obtained from past observations and future predictions can be put to action in order to steer and control the evolution of a software system in a desired direction. Consequently, the field of software evolution is concerned with investigating and understanding the nature of the evolution phenomenon and its underlying drivers, as well as defining engineering principles for achieving software evolution in a controlled manner [107].

At a high level, a lot of activities revolving around understanding, guiding, and supporting software evolution involve five fundamental steps:

- collecting measurements for various characteristics of activities, artifacts, and people,
- keeping records of measurements over time,

- learning from the recorded measurements,
- acting upon what has been learned, and
- measuring the impact of the actions.

The measurements require the definition of adequate characteristics with respect to an assessment task or a set of tasks, and corresponding means for performing the measurements efficiently at a large scale. The historical records require detailed transactional data about activities, artifacts, and people in order to associate the obtained measurements to these entities. The learning requires a problem definition and corresponding data, possibly partitioned and projected over the problem domain, as well as scalable computational infrastructure for processing the data. The actions require an organisational strategy that takes learning outcomes into consideration for future decisions. The assessment of the impact requires adequate evaluation criteria to differentiate and compare the outcomes of different decisions. Data mining and data analytics techniques can be adopted for these purposes and further specialised into software mining and software analytics.

2.2. Data Mining and Data Analytics

Data mining techniques have found widespread application in software evolution studies over the past decade. We consider two fundamental categories of data mining techniques: *directed data mining* for constructing models that explain and/or predict outcomes related to a target characteristic (often referred to as *target variable*); and *undirected data mining* for finding patterns that are not related to a particular target characteristic and whose usefulness is open to human interpretation [110]. Both categories of data mining techniques are used in software assessment for accomplishing various business goals. The choice of a technique depends on the business goal that is to be accomplished and on the given circumstances. Translating the business goal into a data mining task is a fundamental step in the data mining process. The data mining tasks outline the requirements towards the technique, as well as the input and the output data.

When it comes to specifics of the terms data mining and data analytics, there are no universally agreed upon definitions and clear lines between the terms, resulting in their interchangeable use. Both terms have been used as buzzwords at different points in time to denote a broad domain of techniques, approaches, and processes.

Data mining is concerned with extracting and analysing raw data, typically collected for operational purposes, and looking for patterns and relationships. While there may be certain expectations towards the outcome of the data mining process, data mining is often more exploratory in nature. It may also require taking domain knowledge into account in order to guide the evaluation and interpretation of discovered findings.

Data analytics is concerned with analysing data to draw conclusions about what is going on in the subject described by the data, be it a process, an event, or an entity, and guiding decisions or taking actions according to the conclusions. The data is often collected with the specific goal of the analysis and decision making in mind.

2.2.1. Directed Data Mining

Directed data mining techniques rely on the availability of training data with regard to the target characteristic. The target characteristic may be categorical or numerical, where binary categorical characteristics, which are particularly popular as data mining goals, often revolve around modelling binary yes/no questions. Directed data mining techniques are also referred to as supervised learning.

Classification, as one of the most frequently used directed data mining techniques, is concerned with assigning new unknown data items to one of a set of known classes with regard to the values of the target characteristic. The training data containing data items with known classes are used to infer a model which determines the class of a data item based on other characteristics of the data item. The model shall abstract form the training data sufficiently well so that unknown data can still be successfully classified. While classification is concerned with categorical characteristics, estimation is concerned with continuous numerical characteristics. Some classification techniques, such as decision trees [143], logistic regression [31], and neural networks [169], produce an estimate of the probability of the different categories in addition to or instead of the categories themselves. The probability can either be compared against a set threshold or used for ranking. In software engineering, classification and estimation can be used to identify defect-prone artifacts and activities and steer quality assurance efforts [4, 46, 95]. Often times when using directed data mining techniques in software assessment, and in general, the scoring can be considered of primary interest.

In software assessment, and in particular in the classification of e.g. defect-prone vs non-defect-prone artifacts, the outcome is usually concerned with which artifacts should be examined more closely and tested more thoroughly. However, in many cases there is also added value in the ability of the model to explain the classification outcomes. It can provide better understanding of the underlying factors contributing to the classification outcome, such as prominent common characteristics of defect-prone artifacts. Based on this understanding, further measures can be implemented to reduce the prevalence of these characteristics. Some data mining techniques, such as decision trees and logistic regression, are better suited for this purpose than others. The importance of the ability to explain the classification outcomes as well as the characteristics of the input and output data are determining factors in the selection of appropriate technique for a given data mining task, since there are usually trade-offs between explanatory power and accuracy [110].

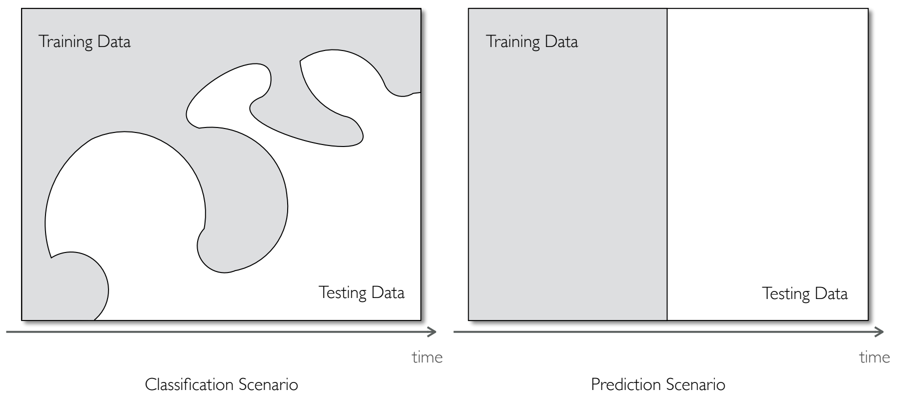


Figure 2.1.: Data partitioning for classification and predictive modelling

If the data includes temporal properties, such as the time of measurement, in directed data mining, there is a further distinction to be made with regard to the temporal relationship between the training and testing data for the model. In *predictive modelling* the testing data comes from a timeframe which strictly after the timeframe of the training data, whereas in *profiling* (or *classification*), the timeframe is the same. Figure 2.1 illustrates both scenarios. In the classification scenario shown on the left, the training and testing data are not temporally separated. In the prediction scenario on the shown on the right, the training and test data are strictly temporally separated.

2.2.2. Undirected Data Mining

While in undirected mining there is no specific target characteristic, there are usually one or more goals that need to be addressed. To address the goal of finding defect-prone software artifacts, we may apply directed data mining if there are available data characterising such software artifacts. If there are no such data, we may rely on undirected data mining to identify software artifacts that appear unusual or dissimilar to the majority of artifacts [197]. Similarly, undirected mining techniques can be applied to identify commonly occurring combinations of characteristics related to the goal in question, or even provide insights for refining the goal or identifying new goals. Undirected mining techniques are also referred to as unsupervised learning.

Clustering is one commonly used undirected data mining technique for segmenting multi-dimensional data into groups exhibiting high similarity, where the interpretation of the clustering results is up to the user. A subsequent examination of the defining characteristics of each cluster can yield new insights regarding common characteristics of the members of each cluster, and possibly also result in new dimensions characterising the original data. New data can then be allocated to the existing clusters or used to continuously refine and redefine the existing clusters. In software assessment, clustering can be used to identify common patterns among developers in order to steer development guidelines and training focus [109, 150]. Clustering can even be used for directed data mining tasks, depending on the similarity of the data items in each cluster with regard to a target characteristic, which determines the ability of the identified clusters to separate the possible values of the target characteristic.

Association rules are another frequently used undirected data mining technique for identifying patterns between co-occurring characteristics. It can be particularly useful for inferring relationships between characteristics of interest, such as ones related to technical risks within individual data points. Association rules can also be applied to multiple (subsequent) data points in order to infer patterns between co-occurring events. This particular scenario is an example of sequential pattern analysis.

2.2.3. Data Preparation

Before the application of data mining techniques, there is often need for processing the data in various ways. This is referred to as data pre-processing or data preparation. This involves the application of data transformation, selection, and data partitioning techniques which are either aligned with business goals, seek to improve some aspect of the data mining process, or help in better understanding the underlying data and guiding the application of subsequent data mining techniques.

Data transformation techniques such as normalisation seek offset any side effects due to high variance in the data ranges for different characteristics (also referred to as "scale-effects"). Normalisation also provides a relative scale view on a data point to be able to better assess where it stands in the context of all other data points.

Information gain [124] with regard to a target characteristic is a main constituent in various data mining techniques such as decisions trees. Information gain can also be used to score the importance of each characteristic in its ability to differentiate data points with regard to the target characteristic. The scores for the characteristics based on their information gain can then be used to rank and filter the most important characteristics with regard to the target characteristic. This is referred to as attribute selection or feature selection. It can help in reducing the dimensionality of the data, simplification of resulting models, as well as reducing over-fitting. The ranking of characteristics can also be used for the characterisation of data sets, in particular in combination with undirected data mining techniques.

Undersampling [42] and oversampling [28, 108] are approaches to balancing data with regard to a target characteristic when the distribution of the target values is not balanced. Strong imbalance in the data used for training in directed data mining may result in biased models that favour one target value over another. Selecting only a subset of data points with the more prevalent value (undersampling) or synthetically producing more data points with the less prevalent value (oversampling) during training can improve the performance of the applied technique by reducing inherent bias. However, undersampling and oversampling may also have the opposite effect if the testing data is also following a distribution skewed towards one target value.

Weighting adds meta information to each data point which may be taken into consideration by a data mining technique in order to place higher importance on particular data points. Weighting can be used to offset some of the effects of imbalanced data sets, where data points having the minority target value are assigned higher weight, inversely proportional to their ratio within the data set. Correspondingly, data points having the majority target value are assigned a lower weight, also inversely proportional to their ratio within the data set.

2.3. Mining Software Repositories

Software development produces rich data sets containing operational data related to different activities by various stakeholders, which are scattered across multiple assets. Software repositories, such as a VCS and an ITS, are common kinds of assets which containing detailed information related to the evolution of a software system. However, the data stored in VCS and ITS is primarily intended for the operational needs of such systems and not necessarily for the purposes of understanding software evolution. Hence, such assets need to be processed in order to extract different kinds facts relevant for the understanding of software evolution phenomena and for aiding decision making. Adequate tools, techniques, and skills are necessary to make the most out of the potentially powerful information sources. The field of *Mining Software Repositories* (MSR) explores different approaches for the systematic extraction of information from software repositories both in the form of basic facts (measurable information) and in the form of derived knowledge (actionable insights) [20].

2.3.1. Data Sources, Metrics, and Facts

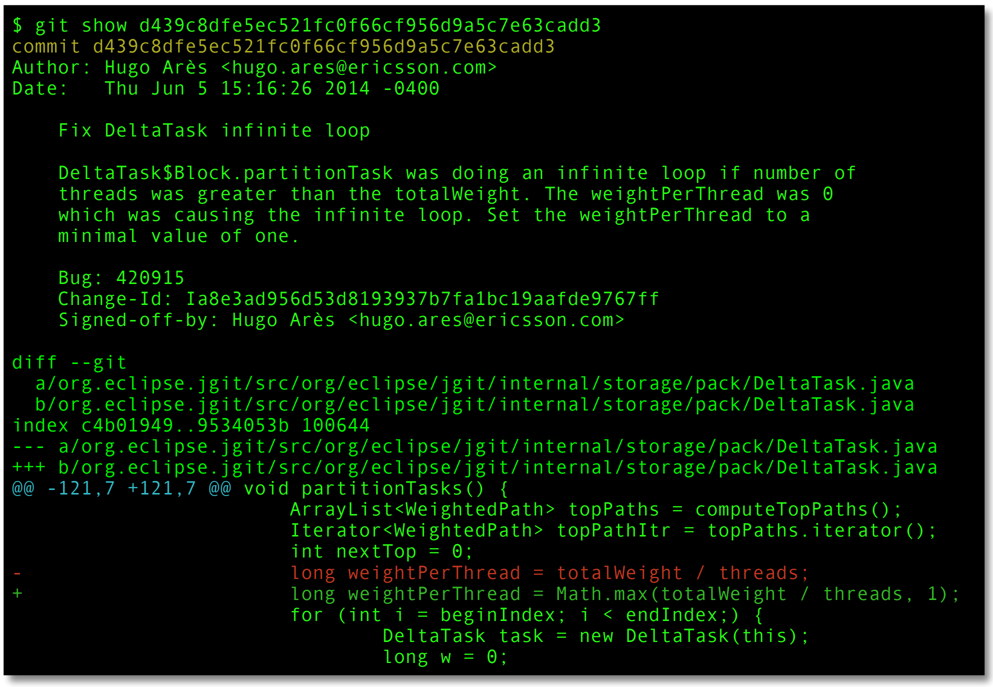
*Version Control System*s (VCSs) are used to keep track of incremental changes in files. In the earliest form, VCSs were focused on providing means to record successive versions of files and to revert to earlier versions if necessary. In addition to individual versions of the files, meta-data regarding the time, purpose, and person responsible for the version was added in order to make navigating the different versions easier. Later on, text-based differencing algorithms were integrated in VCSs in order to reduce redundancy due to mostly unchanged parts of files and make the storage of different versions more efficient. As an added benefit, it became possible to identify and navigate individual changes within textual files, rather than contemplating a version of a file as a monolithic unit. Changes to multiple files are often related, so it is often necessary to record such changes as *change sets*. Change sets are particularly important in managing changes to software systems as a functional addition to a software such as a new feature often involves changes to multiple files. Thus, a change set can record all the changes related to a particular feature and enable a user to restore the software system to a state before the feature was introduced with a single action, rather than having to restore each relevant file separately to a corresponding state where the system is still consistent. Such approaches to manage sets of changes were inherited from well established practices in industrial design and manufacturing, where for different models and prototypes of machines, different versions of individual components were required. In an environment where components are manufactured and assembled independently, a disciplined approach to managing the different versions of related components and their corresponding designs were essential.

Along with changes in the ways people work with files within a VCS, the evolution of VCSs also brought changes to the ways people work with other people. Server-based and networked VCSs enabled multiple people to collaborate and share the same centralised code-base by checking in their changes and checking out changes from other people. With added benefits of centralised networked VCSs such as *Subversion*[5], there were still some limitations, such as the need to have network connection in order to store changes. To overcome some of the limitations, distributed VCSs, such as *Git*[6] emerged, where everyone working on a project would have a local copy of the complete history of a project or part of a project. Changes can then be exchanged among all involved people and gradually integrated in the end result in a distributed manner.

As far as terminology is concerned, various terms are used to refer to similar notions, where *revision*, *version*, *change set*, *commit*, *state* are often used interchangeably in the context of VCSs[7]. Correspondingly, VCSs are also sometimes referred as revision control, source control, or configuration management systems (although the latter has a broader meaning). Changes between successive versions of a file are referred to as *patches*, *diffs* (short for "difference"), or *deltas*. File comparison tools, such as *GNU Diffutils* [8], operate at the line level of granularity where lines in a file are compared and if they do not match, the corresponding lines are either added, removed, or replaced (where a replacement is a combination of removing a line and adding a new line at the same place). Such functionality is integral in VCSs. More sophisticated approaches such as *LSDiff* [48, 93] and *ldiff* [23] provide more detailed and refined comparison.

While there are various *Graphical User Interface* (GUI) tools for working with VCSs, most VCSs provide *Command Line Interface* (CLI) shells for accessing their functionalities, as well as files and meta-data stored in them. An example showing the

[5]See `https://subversion.apache.org`
[6]See `https://git-scm.com`
[7]See `https://git-scm.com/docs/gitglossary` for example
[8]See `http://www.gnu.org/software/diffutils/`



```
$ git show d439c8dfe5ec521fc0f66cf956d9a5c7e63cadd3
commit d439c8dfe5ec521fc0f66cf956d9a5c7e63cadd3
Author: Hugo Arès <hugo.ares@ericsson.com>
Date:   Thu Jun 5 15:16:26 2014 -0400

    Fix DeltaTask infinite loop

    DeltaTask$Block.partitionTask was doing an infinite loop if number of
    threads was greater than the totalWeight. The weightPerThread was 0
    which was causing the infinite loop. Set the weightPerThread to a
    minimal value of one.

    Bug: 420915
    Change-Id: Ia8e3ad956d53d8193937b7fa1bc19aafde9767ff
    Signed-off-by: Hugo Arès <hugo.ares@ericsson.com>

diff --git
  a/org.eclipse.jgit/src/org/eclipse/jgit/internal/storage/pack/DeltaTask.java
  b/org.eclipse.jgit/src/org/eclipse/jgit/internal/storage/pack/DeltaTask.java
index c4b01949..9534053b 100644
--- a/org.eclipse.jgit/src/org/eclipse/jgit/internal/storage/pack/DeltaTask.java
+++ b/org.eclipse.jgit/src/org/eclipse/jgit/internal/storage/pack/DeltaTask.java
@@ -121,7 +121,7 @@ void partitionTasks() {
                        ArrayList<WeightedPath> topPaths = computeTopPaths();
                        Iterator<WeightedPath> topPathItr = topPaths.iterator();
                        int nextTop = 0;
-                       long weightPerThread = totalWeight / threads;
+                       long weightPerThread = Math.max(totalWeight / threads, 1);
                        for (int i = beginIndex; i < endIndex;) {
                                DeltaTask task = new DeltaTask(this);
                                long w = 0;
```

Figure 2.2.: Git VCS example from the jgit project

information related to a change in the *jgit* project accessed through the CLI shell is shown in Figure 2.2. The example includes:

- the identifier for the change (*d439c8..add3*),
- the author (*Hugo Arès*) and their e-mail address,
- the date and time of the change (*15:16 on Jun 5, 2014*),
- a textual description of the change (*Fix DeltaTask* ...),
- the files affected by the change (*DeltaTask.java*),
- the location and length of the changes within the files (*@@ -121,7 +121,7 @@*),
- as well as the context and content of the changes.

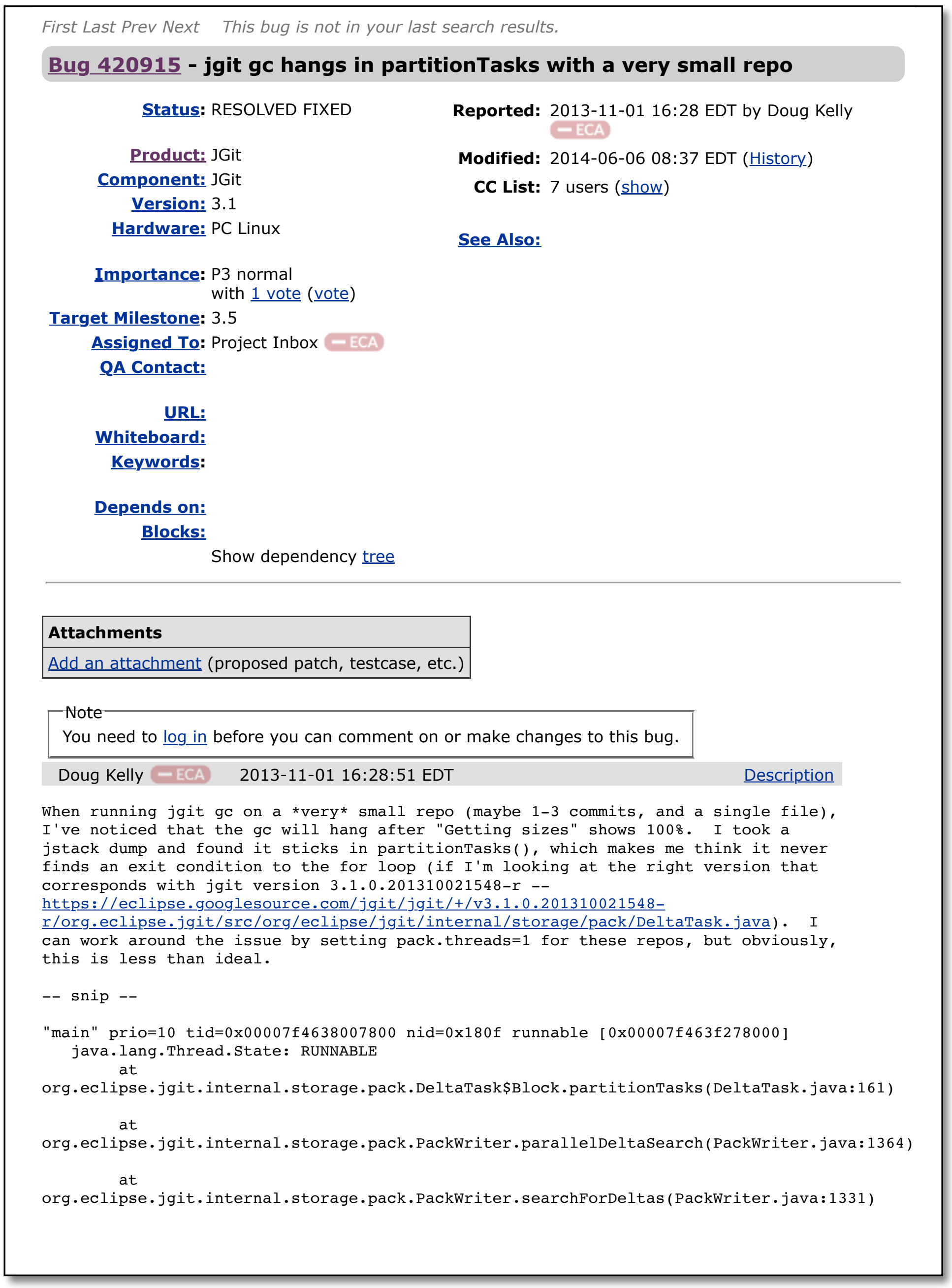
The content of the changes is typically shown as lines that have been removed (indicated by a "-" prefix) and lines that have been added (indicated by a "+" prefix) presented in the context of (typically three) unchanged lines before and after each change. The exact location of the changed lines (excluding the context) can be computed by using the information regarding the location of the shown context (starting at line 121) and the prefixes for indicating the changed lines, resulting in line 124.

The textual description of the change may contain further structured information, including who reviewed or signed off the change, as well references to other sources of information, such as review management platforms and *Issue Tracking System*s (ITSs). ITSs, also commonly referred to as *bug databases* in the literature, are used for reporting and managing issues related to products. We prefer the more neutral terms *issue* and ITS since not all reported issues are bugs as requests for new features or other kinds of changes are also reported as issues. In some projects issue tracking systems are also used more broadly for task coordination. Issues in ITSs typically contain an identifier, both a short and long description of the issue, name and e-mail of the person who reported the issue, the date and time the issue was reported, as well as its current status. Incoming issues are typically recorded with the status *new*, then they may be *assigned* to someone, and finally *resolved* in one way or another. There may be additional stages in between depending on the established processes within a project.

Additional meta-data may indicate the product and component the issue is concerning, related issues and the nature of the relations, as well the version in which it was discovered and possibly the version in which it was addressed or the target version in which it will be addressed. Targeted version or milestone in an ITS is one way of recording process information regarding development and release milestones, providing additional context for development activities. Some information regarding milestones is also available in VCSs in the form of *tags*. Depending on the project needs, even more meta-data may be recorded through further customisations of the ITS. Some ITSs provide means for discussing an issue so that further details describing the issue can be requested and possible solutions can be provided and tested before the issue is marked as resolved, or at the very least keep track of people that need or want to be informed when there are changes to an issue. Finally, attachment may be provided as part of the description or as part of the solution for an issue. Modern ITSs also keep track of detailed information regarding all modifications to the issue.

The identifier of an issue is typically used for referencing the issue in other systems such as VCSs. The issue referenced in description of the change shown in the example in Figure 2.2 ("*BUG: 420915*") is illustrated in Figure 2.3 (excerpt, the complete description and comments are not shown). In this example, we can observe that the issue was reported on *2013-11-01* and last modified on *2014-06-06*. A look at the detailed event history shown in Figure 2.4 confirms that indeed the issue was resolved on the later date. From this example we can conclude that it took around seven months to process that issue. We also observe that the ITS was updated about one day after the corresponding change was recorded in the VCS.

In the examples discussed so far, there were explicit references from one data source (VCS) to another (ITS). This makes linking activities in both systems both for operational and for research purposes rather easy, where information regarding the purpose or impact of a change can be enriched with additional information from the ITS. However, linking is not always straightforward, especially when it comes to merging different



First Last Prev Next This bug is not in your last search results.

Bug 420915 - jgit gc hangs in partitionTasks with a very small repo

Status: RESOLVED FIXED
Reported: 2013-11-01 16:28 EDT by Doug Kelly ECA
Product: JGit
Modified: 2014-06-06 08:37 EDT (History)
Component: JGit
CC List: 7 users (show)
Version: 3.1
Hardware: PC Linux
See Also:
Importance: P3 normal with 1 vote (vote)
Target Milestone: 3.5
Assigned To: Project Inbox ECA
QA Contact:
URL:
Whiteboard:
Keywords:
Depends on:
Blocks:
Show dependency tree

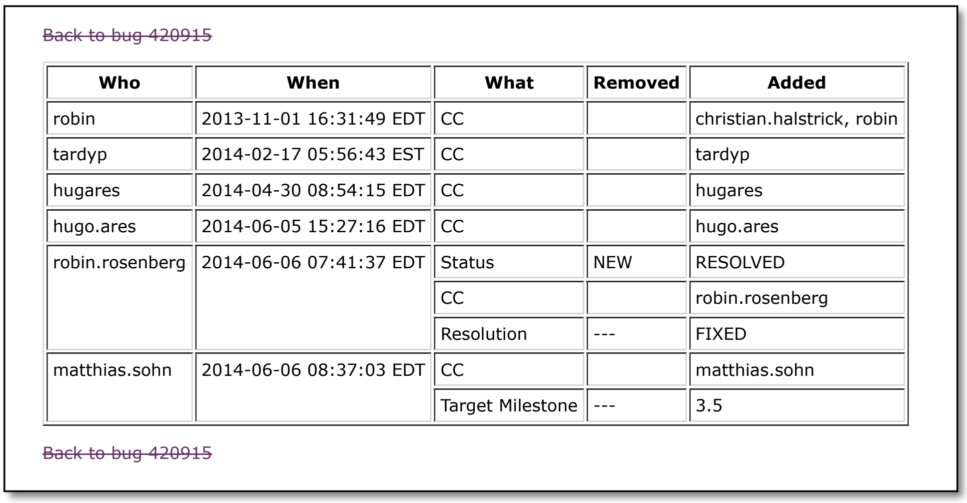
Attachments
Add an attachment (proposed patch, testcase, etc.)

Note
You need to log in before you can comment on or make changes to this bug.

Doug Kelly ECA 2013-11-01 16:28:51 EDT Description

```
When running jgit gc on a *very* small repo (maybe 1-3 commits, and a single file),
I've noticed that the gc will hang after "Getting sizes" shows 100%.  I took a
jstack dump and found it sticks in partitionTasks(), which makes me think it never
finds an exit condition to the for loop (if I'm looking at the right version that
corresponds with jgit version 3.1.0.201310021548-r --
https://eclipse.googlesource.com/jgit/jgit/+/v3.1.0.201310021548-
r/org.eclipse.jgit/src/org/eclipse/jgit/internal/storage/pack/DeltaTask.java).  I
can work around the issue by setting pack.threads=1 for these repos, but obviously,
this is less than ideal.

-- snip --

"main" prio=10 tid=0x00007f4638007800 nid=0x180f runnable [0x00007f463f278000]
   java.lang.Thread.State: RUNNABLE
        at
org.eclipse.jgit.internal.storage.pack.DeltaTask$Block.partitionTasks(DeltaTask.java:161)

        at
org.eclipse.jgit.internal.storage.pack.PackWriter.parallelDeltaSearch(PackWriter.java:1364)

        at
org.eclipse.jgit.internal.storage.pack.PackWriter.searchForDeltas(PackWriter.java:1331)
```

Figure 2.3.: Bugzilla ITS example from the jgit project



~~Back to bug 420915~~

Who	When	What	Removed	Added
robin	2013-11-01 16:31:49 EDT	CC		christian.halstrick, robin
tardyp	2014-02-17 05:56:43 EST	CC		tardyp
hugares	2014-04-30 08:54:15 EDT	CC		hugares
hugo.ares	2014-06-05 15:27:16 EDT	CC		hugo.ares
robin.rosenberg	2014-06-06 07:41:37 EDT	Status	NEW	RESOLVED
		CC		robin.rosenberg
		Resolution	---	FIXED
matthias.sohn	2014-06-06 08:37:03 EDT	CC		matthias.sohn
		Target Milestone	---	3.5

~~Back to bug 420915~~

Figure 2.4.: Bugzilla ITS events example from the jgit project

identities of the same developer across multiple systems, or when there is no explicit information provided in either system [152, 180].

Apart from VCSs and ITSs, mailing lists and user forums are other sources of information regarding the evolution of a software system. Mailing lists typically contain information regarding communication among developers whereas user forums provide more information regarding communication among users. However, often projects also setup mailing lists for users, and on the other hand developers may be involved in discussions on user forums by providing further support or information to users. Similar to linking activities between VCSs and ITSs, links between these systems and mailing lists and/or user forums can be established as well in order to keep track of developer activity and interactions on mailing lasts or user forums, for example.

Based on the information stored in the different data sources, various metrics can be calculated to obtain quantitative measurements regarding the state of a software system at a given point in time as well as regarding how it got to that state over time. These include the number and size of changes applied to a file, the frequency of changes to a file, or the number of developers working on a file, for example, or also more sophisticated measurements [54, 129]. Such metrics have been used in software evolution research in various application scenarios, such as finding refactorings [38] and predicting defects [69, 100, 126, 129]. Information regarding collaboration among developers based on social and other metrics has been similarly used for various applications [37, 40, 123, 172, 188, 201].

Besides exploiting the meta-data stored in VCSs and ITSs, a more traditional and common approach to measurement in software development is to quantify different characteristics of the actual files in the VCSs. Such characteristics include various size and complexity metrics [30, 119]. Approaches such as the *Goal Question Metric* (GQM) approach [8] seek to make the application of software metrics more systematic and goal-oriented. Traditional size and complexity metrics are widely used in software engineering research and practice, including effort estimation [1, 158], evaluate maintainability [146] and reliability [160], and predict defects [4, 46, 122, 128, 139]. Such measurements can only be performed on structured text files such as source code files, provided they are in a programming language for which means for performing such measurements are available. In addition, available measurements may vary from programming language to programming language.

Computing size and complexity metrics falls under the broader category of static analysis. Other types of static analysis include detecting code smells [50], as well as detecting duplicated code [9] as a particular type of smell with considerable implications on the evolution of software systems [6, 43, 94]. Static analysis is also useful for identifying and tracking the evolution of logical entities such as classes and methods, as well as dependencies among them. Various tools supporting different types of static analysis, both standalone, and integrated in *Integrated Development Environment*s (IDEs) and *Continuous Integration* (CI) platforms have already found widespread use.

All of the characteristics regarding the state of a software system and its evolution up to that state can be generalised as basic *facts* — observations and evidence regarding the state of affairs concerning a software system at a given point in time. Any additional information obtained by assessing these facts we will refer to as *derived knowledge*. Since software systems continuously evolve, so do facts and derived knowledge.

Operational systems such as VCSs and ITSs are designed for a specific purpose, serving users along intended usage scenarios. They produce and maintain data as needed for the intended purposes. As a consequence the data is not necessarily easily accessible or in a suitable format for mining. Related data often is spread across multiple operational systems, each serving its designated purpose and relationships between data from the different systems are often only implicit. A big part of software mining involves extracting, organising, and integrating software development data from different sources. While the available data can be considered truthful, in that it is not biased as a result of the measurements, since at the time it is produced, measurements are likely not considered, it can also be considered often rather noisy, possibly incomplete, and even incorrect. The availability and quality of data are often cited as limiting factors in software mining and its adoption by practitioners [65, 142, 162].

2.3.2. Change Labelling and Classification

Determining and understanding the purpose and consequences of changes is central topic in software assessment. It can help in characterising the work performed on evolving systems as well as in better understanding and potentially improving the maintenance and evolution process.

Evolutionary annotations [55] describe how code evolves over time by capturing rationale and intent of changes based on various indicators, such as change logs, VCS and ITS data, mailing lists, code comments, etc. They aim to provide a framework for the analysis and understanding of changes by means of enriched meta-data describing changes, patterns, and activities related to the software development process and software evolution, based on directly and indirectly measurable change-related properties. This, in turn, enables qualitative assessment of the impact and risks associated with making certain types of changes.

The simplest way to classify changes is by their size [68]. Literature and practice indicate, for example, that large changes are indicative of code management, re-engineering, and corrective engineering tasks, such as formatting, documentation, refactoring, and bug fixing. Small changes on the other hand are indicative of development and forward engineering tasks such as the implementation of new features.

Another common and simplistic approach to identifying the purpose of changes based on keywords [68]. While VCSs record information regarding the purpose of a change. However, this information is usually provided in informal free text. While there may be some (semi-) structured parts within it, those can be difficult to identify consistently. One common approach is to look for references to issues in an ITS and then look up relevant information from the ITS. However, the ITS, while providing some additional meta-data, still relies mostly on unstructured textual information. Looking designated keywords in both the description of referenced issue in the ITS, as well as the description of the change in the VCS, can provide further clues as to what the intended purpose of the change really is.

More sophisticated approaches consider changes to the structure, signatures and dependencies of logical elements within code files in order to identify refactorings [96, 183], provided that static analysis is available. Considering changes in the broader context of other changes enables the identification of behavioural patterns [57] of interactions between developers. The changes can then be described by the role they play within these patterns.

Apart from investigating the consequences of changes, approaches based on origin analysis [58, 59] and line tracking [22, 97, 117] seek to identify causes for consequences in subsequent changes [165]. Such approaches typically identify when a piece of code within a file that is considered to be fixing some problem was last changed and consider the corresponding change as the cause for the problem.

Further ways to describe changes can be based on power consumption [78] which is particularly relevant for power constrained devices but may also correlate with other change attributes, such as invocation of error handling routines due to introduced faults. Test-related change patterns [194] can also be used to describe the type of changes taking place, where increase in test code without corresponding increase in production code can be indicative in testing refinement or other test-related changes.

Due to the nature of how VCSs operate and in particular how people work, we need to consider the possibility of tangled changes [76, 148]. Tangled change sets incorporate changes exhibiting different characteristics individually, but when collapsed together may be difficult to discern in any meaningful way, despite being seemingly natural from a developer's point of view.

All of the approaches discussed in this section are concerned with identifying the intent of changes based on direct indicators when it comes to consequences. But when it comes to the causes for these consequences, these can only be identified retrospectively once the consequences have been identified. Data mining techniques used in software assessment aim to identify other characteristics that are associated changes that cause certain consequences. A common application of data mining techniques in software assessment is defect prediction.

2.3.3. Defect Prediction and Software Analytics

The extraction of basic facts can be considered well established in software engineering research and practice. Making sense of all the extracted facts is still a non-trivial task.

Defect prediction [4, 65, 95] is the application of data mining techniques to data related to software development with the purpose of identifying parts of the software or changes to the software that are particularly susceptible to defects based on models extrapolated from available data. The available data is typically in the form of various facts. The models are extrapolated from the facts by applying directed data mining techniques in order determine which characteristics of the code correlate with high defect likelihood. The extrapolated models produce predictions indicating the most likely suspects with regard to defects. The predictions, if reliable, may be used to make predictions about new data and guide quality assurance efforts in order aid practitioners in different decision scenarios, such as targeting review, testing, and refactoring. In order to obtain the prediction models, sufficient training data containing also information regarding the target variable (defective or not) shall be available.

Related to software mining is the emerging field of software analytics [195], where analysis, data, and systematic reasoning are applied in a layered manner to obtain insightful and actionable information to aid in software development decision making. In [20] and [21], the authors surveyed professional software engineers about their information needs with regard to data-driven decision making. While MSR can be considered a more data-focused approach, in [20] and [21] the authors call for software analytics to pursue a more user-focused approach, where a user is to be taken as the starting point and the relevant data is determined based on their specific needs and on the problems they need solutions for. The authors organised a set of analysis types that can address such needs by time frame (past, present, future) and category of technique (exploration, analysis, and experimentation):

- Trends analyses such as regression analysis quantify how artifacts are changing.
- Anomaly detection can raise alerts on unusual events.
- Forecasting through extrapolation can serve as an indication for future events based on trends.
- Summarisation through topic analysis can characterise key aspects of artifacts.
- Different views on data in interactive overlays and correlations can help in manual analysis.
- Including goals in analyses provides assistance for planning can serve in impact analysis with respect to the goals.
- Machine learning can be used to model normal development behaviour
- Benchmarking can be used to compare the outcomes of different practices and decisions.
- Simulation can be used to explore and test the outcomes of different decisions in the future.

To address such specific needs and problems, the authors extrapolated from their study that analytics tools shall be easy to use, fast, and produce concise output, while supporting different types of artifacts and indicators, and enabling drill-down into data based on different perspectives. With the increasing attention to human-related factors, software mining and software analytics can also be used to aid team organisation, training, and knowledge management decisions in software development.

While it is often performed retrospectively, there is a growing recognition that software analytics should be applied as an ongoing process, integrated into the software development process [113]. This integration allows software analytics to feed from data produced during the software development and provide actionable insights based on multi-faceted analyses. Measurement of the impact of the actions undertaken based on the insights from the software analytics in turn produces new data. This data are fed into the software analytics again to assess their impact in the last round, generating new insights and so on, resulting in a cyclic ongoing process. Such an integration requires alignment of development and business strategies in order to include continuous application of software analytics, allocating necessary resources for it, and determining adequate ways to act on the insights resulting from the software analytics. The necessary resources include data analysts that can define the problem in data mining terms and understand the results from the analytics, rather than concentrating on the application of readily available facilities without considering the peculiarities of the specific application context.

As with all data mining in general, software mining models also need to evolve, otherwise they become obsolete [173]. Considering that software mining models are software systems, the principles of software evolution apply to them as well. As the environment in which these models are used continuously changes, with new data reflecting the changing circumstances which software systems are developed becomes available, old data may become less and less useful over time.

Mining results can be used for recommendation systems to provide personalised feedback to developers when they perform activities under particular circumstances. Software mining can also been used to prioritise certain development and organisational activities, as well as adjusting human resources and team structures. Ultimately, the acceptance of various approaches for getting actionable insights and putting them into practice is still called into question [142].

2.4. Modelling and Model Based Engineering

Modelling and related terms have a broad variety of meanings depending on the context in which they are being used. Every scientific and engineering practice employs some sort of modelling. Even daily activities often rely on modelling of various aspects of every day life.

2.4.1. Modelling and Meta-modelling

Modelling, in broader terms, involves any activity that results in creating a representation of a subject with certain goals in mind. The representation is abstract and simplified in that it does not include characteristics of the subject that are not relevant to the goals for which the model was created. The goals typically include reasoning about the subject and answering specific questions regarding the subject [13]. Thus, a model shall support reasoning about the subject and for it to be useful, the model shall make reasoning and answering questions regarding the subject easier in comparison to using the actual subject for the same purposes.

Applying the same reasoning to modelling, meta-modelling involves creating a representation of what can be a part of a model. This step is also referred to as domain modelling. It involves identifying the constituents of a model including relevant concepts and relationships among them that are specific to the domain. Taking this a step further leads to meta-meta-modelling which determines the necessary constituents of meta-models.

The *Unified Modelling Language* (UML) [132] standardised by the *Object Management Group* (OMG) has found widespread use in industry and academia. It aims to address modelling needs by a set of concepts and representations for various uses at different levels of abstraction. The UML architecture shown in Figure 2.5 describes four layers reflecting the different abstraction levels noted above.

The *M3* layer defines the fundamental concepts necessary for the specification of meta-models. The *Meta-Object Facility* (MOF) [134] can serve as the high-level foundation in this architecture. The *M2* layer is where the meta-model for the domain is

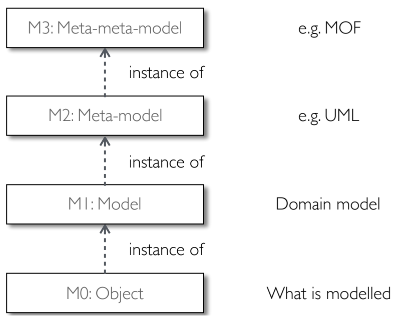


Figure 2.5.: UML architecture

defined. This can be the UML meta-model for the domain of modelling languages including UML itself or a custom meta-model for different domain. The *M1* layer is where instances of the meta-model are defined. For UML, this includes UML models defined by users of the UML. For other domains, this includes models that are instances of the corresponding domain models defined in *M2*. Finally, the *M0* layer is where the real-world objects described by the models are.

The meta-models defined using MOF are represented by using a subset of the UML *Class Diagram*, where relevant concepts are defined as *Classes* and relationships between them are defined as *Associations*. Further constraints can be applied on top of the meta-model by means of languages such as the *Object Constraint Language* (OCL) [133]. OCL is declarative language for the specification of rules and expressions over meta-models based on MOF.

An example of the basic constituents of a meta-model is shown in Figure 2.6. The meta-model in this example is concerned with the domain of user interfaces consisting of toolbars containing one or more buttons, where each button has a label and triggers a functionality. Using the notation of the UML *Class Diagram*, the corresponding concepts for toolbars, buttons, and functionalities are represented as (meta-) classes, with the relationships among them represented as associations. The associations are quantified by means of multiplicities next to the association name. The classes may also contain attributes such as the *label* attribute for the *Button* class. An instance of this meta-model will contain one or more toolbar elements, each toolbar containing one or more buttons, where each button refers to one functionality that can be triggered by pressing the button. At this point, this representation is still very abstract and simplified, omitting any information regarding the size and color of the buttons for example, or even how exactly the buttons shall be interacted with and how the corresponding functionality shall be triggered as a result. The actual buttons in an implementation of the modelled user interface are the real objects that are subjected to modelling. Further restrictions, such as limiting the size of the label in a button to 10 characters, may also be specified by means of OCL as shown in the example in Listing 2.1.

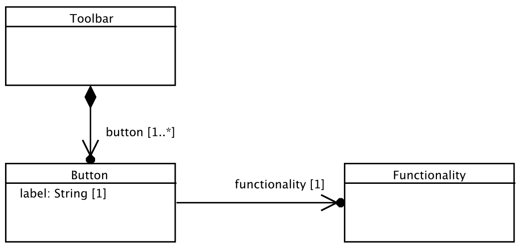


Figure 2.6.: Meta-modelling example

```
context Button
    inv: self.name.size() <= 10
```

Listing 2.1: OCL constraint example

Even such a simplified model, enables reasoning about questions such as whether all available functionalities are accessible via buttons. One way to pose such questions is by means of constraints. Listing 2.2 illustrates the constraint of having at least one button that triggers a given functionality.

Modelling frameworks and related supporting technologies enable the validation, storage, visualization, and transformation of all model instances conforming to meta-models described by using a given modelling framework. The *Eclipse Modeling Framework* (EMF)[9] [19] provides an implementation of OMG's *Essential Meta-Object Facility* (EMOF) supplying the necessary facilities in order to enable pragmatic realisation of modelling and meta-modelling tasks.

2.4.2. Model Based Engineering

Model Based Engineering (MBE), in the broadest sense, includes any engineering practice or process that makes use of modelling, where models may also play a supporting rather than central role in the process. Models may be used only for design and communication, but are not processed automatically to produce building blocks of the end product. While they are considered important, they are not the basis for the development process.

Model Driven Engineering (MDE) is a subset of MBE where models play a central role and are thus considered part of the principal output of the engineering process. The actual implementation is then partially or completely generated from the models by automated means. In MDE, models may also be used beyond the development activities, including supporting the evolution of the system or also reverse engineering legacy sys-

[9]See `http://www.eclipse.org/modeling/emf/`

```
context Functionality
    inv: Button.allInstances()>exists(b | b.functionality = self)
```

Listing 2.2: Using OCL to pose questions

tems. In contrast, the scope of *Model Driven Development* (MDD) is slightly narrower, where models are primarily used for development in order to streamline the implementation of repetitive standard tasks that are frequent source of errors and additional overhead. Consequently, models become (part of) the implementation of a system. The *Model Driven Architecture* (MDA) [164] approach standardised by the OMG takes this further, envisioning the use of sufficiently sophisticated models that can be automatically transformed through various abstractions from computation independent through platform independent to platform specific models targeting different deployment platforms by means of corresponding tools. As an OMG standard, MDA is focused on the application of UML and other OMG standards in MDD.

Traditionally there has been an assumption that models are created during analysis and design and then frozen during implementation. Advancements in technology and adoption among agile software development practitioners have shown that MDD can be applied iteratively as well and, in particular, that modelling can be applied during implementation as well. While there is still an assumption that the models are sufficiently detailed to be useful during implementation, *Agile Model Driven Development* (AMDD) seeks to drop that assumption as well, and take the middle way by focusing on agile models that are “barely good enough” [2] and letting them evolve incrementally and iteratively just in time during the development. *Domain-Specific Modeling* (DSM) [89] is a related approach to dealing with modelling complexity by raising the level of abstraction while narrowing the focus of the modelling activities by using concepts from the target problem domain rather than higher level concepts that are still tied to generic programming notions. By using domain specific code generators it enables full code generation tailored to the target domain.

2.4.3. Model Translation and Transformation

Models describe higher level abstract representations. To make use of these abstract representations in practice, they need to be stored and accessed by the various modelling tools and platforms such as the EMF. The EMF relies on *Resource* implementations containing serialisations of model instances. The *XML Metadata Interchange* (XMI) [135] standard by the OMG defines a simple way to represent abstract models in the form of *Extensible Markup Language* (XML) documents. The EMF also relies on XMI as the standard *Resource* implementation for the serialisation and de-serialisation of model instances. Beyond XMI, the EMF also provides support for a binary *Resource* implementation which has certain benefits with regard to space and memory

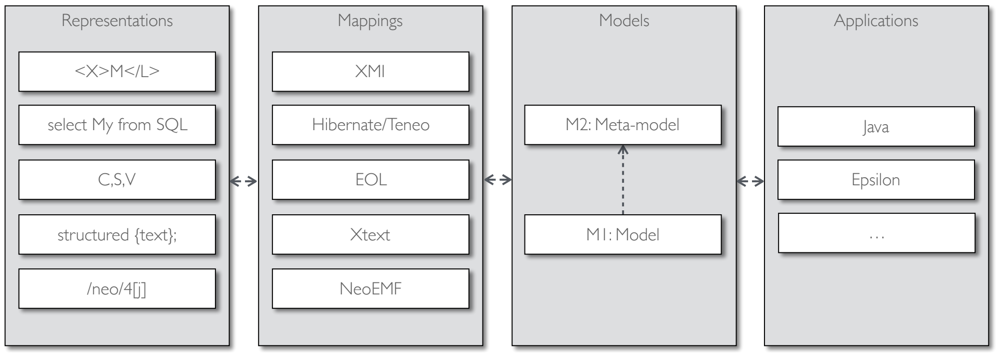


Figure 2.7.: Representations to applications

requirements at the expense of certain compatibility constraints. Other concrete representations, such as textual documents, relational databases, and NoSQL databases, can be mapped to a given meta-model by means of corresponding *Resource* implementations. This enables both the reuse of existing (legacy) data representations, as well as the translation of model instances from one concrete representation to to accommodate different usage scenarios, while still relying on a common underlying information model.

Figure 2.7 illustrates the relationships between different applications relying on a common underlying meta-model, the instances of which may be stored in various concrete representations. Given a meta-model, a concrete representation of an instance of that meta-model, and a corresponding mapping for the type of concrete representation to the meta-model in the form or a *Resource* implementation, an application can access and modify the instance transparently regardless of its concrete representation. Similarly, a given concrete representation of a model can be seamlessly translated to another concrete representation provided corresponding mappings are available.

Consider an example instantiation of the user interface meta-model illustrated in Figure 2.8 represented as an UML object diagram. In the given example, there is one *Toolbar* instance `mainToolbar` containing three *Button* instances, `open`, `close`, and `quit`, with corresponding labels and relationships to corresponding functionalities. Two different concrete representations for this model instance are shown in Figure 2.9. On the bottom left, an XMI representation in the form of an XMI *Resource* according to the generic mappings in the XMI standard is shown. On the bottom right, a textual representation in the form of an *Xtext*[10] *Resource* according to a customised Xtext mapping is shown. Xtext is a framework for specification of textual representation of models. It provides further facilities targeted towards the development of model-based programming languages and domain-specific languages in particular. Xtext mappings are spec-

[10]See `https://eclipse.org/Xtext/`

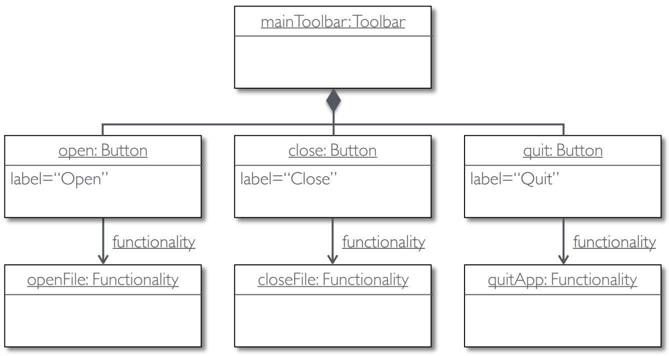


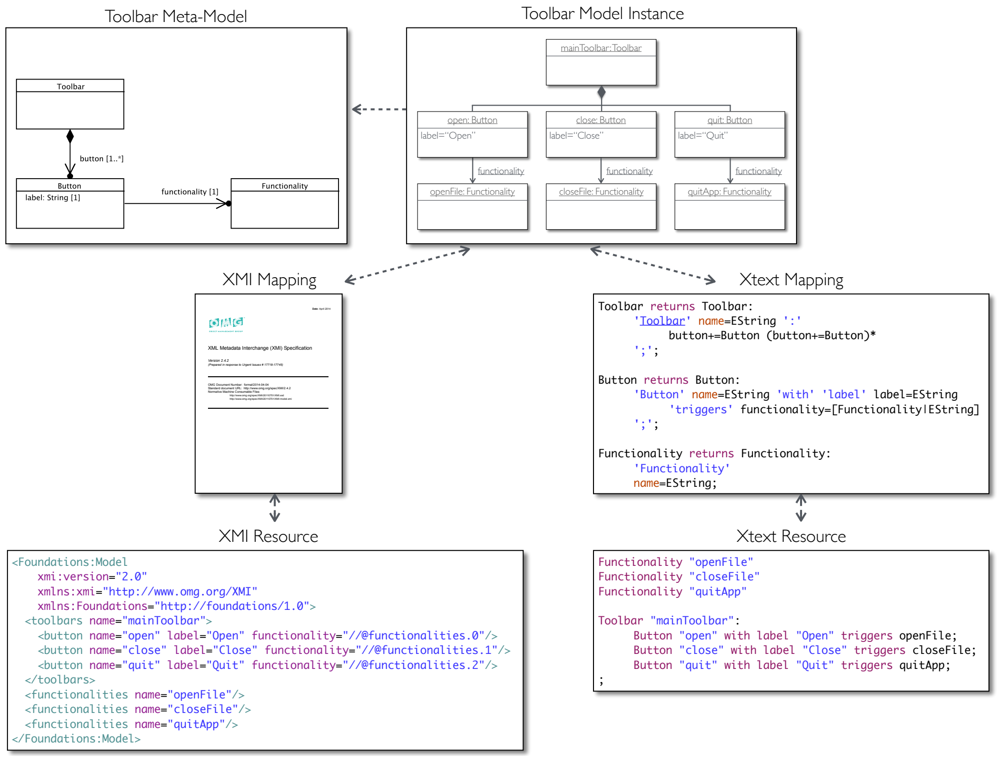
Figure 2.8.: Object diagram for toolbar model instance

ified in the form of annotated *Extended Backus–Naur Form* (EBNF) grammars. The annotations indicate how the textual representations are related to corresponding meta-model elements. For example the text `Functionality "openFile"` is mapped to an instance of the *Functionality* meta-class with the same name. Conforming to the same meta-model, both representations can be used interchangeably and translated into one another. A similar approach is pursued to provide mappings for relational databases by means of Hibernate and Teneo[11]. While both Xtext and Hibernate/Teneo require custom mappings (default ones can be automatically generated), NeoEMF[12] relies on a more generic approach, similar to XMI where the user has no influence on how the meta-model is mapped to the NoSQL database.

Beyond translation between different resources for the same model instance, model transformations allow existing models to be transformed to new models or enriched with further information at a higher level of abstraction. Model transformations are typically specified in the form of rules defining relationships between elements of the corresponding meta-models. Figure 2.10 summarises the idea behind *Model-to-Model* (M2M) transformation. Given a source meta-model and a target meta-model, a set of transformation rules specified for mapping concepts from the source meta-model to concepts of the target model can be used to automatically obtain target model instances corresponding to existing source model instances. Worth mentioning is that the source and target meta-models and corresponding model instances need not necessarily be different. M2M transformation can also be performed on the same model to transform or enrich the model with additional information (sometimes referred to as endogenous transformation). Additionally, there may be multiple source and target models involved in a transformation specification.

Various model transformation technologies have been developed over the past

[11]See `https://wiki.eclipse.org/Teneo/Hibernate`
[12]See `http://www.neoemf.com`



```
Toolbar returns Toolbar:
      'Toolbar' name=EString ':'
            button+=Button (button+=Button)*
      ';';

Button returns Button:
      'Button' name=EString 'with' 'label' label=EString
            'triggers' functionality=[Functionality|EString]
      ';';

Functionality returns Functionality:
      'Functionality'
      name=EString;
```

```
<Foundations:Model
    xmi:version="2.0"
    xmlns:xmi="http://www.omg.org/XMI"
    xmlns:Foundations="http://foundations/1.0">
  <toolbars name="mainToolbar">
    <button name="open" label="Open" functionality="//@functionalities.0"/>
    <button name="close" label="Close" functionality="//@functionalities.1"/>
    <button name="quit" label="Quit" functionality="//@functionalities.2"/>
  </toolbars>
  <functionalities name="openFile"/>
  <functionalities name="closeFile"/>
  <functionalities name="quitApp"/>
</Foundations:Model>
```

```
Functionality "openFile"
Functionality "closeFile"
Functionality "quitApp"

Toolbar "mainToolbar":
      Button "open" with label "Open" triggers openFile;
      Button "close" with label "Close" triggers closeFile;
      Button "quit" with label "Quit" triggers quitApp;
;
```

Figure 2.9.: Different concrete representations for a toolbar model instance

decades. A set of languages collectively known as *Query / View / Transformation* (QVT) [136], which includes the imperative *QVT-Operational* and declarative *QVT-Relations*, is standardised by the OMG. The Epsilon family of languages and tools for working with EMF-based models providing facilities for various tasks, such as code generation, M2M transformation, validation. Of particular interest are the *Epsilon Object Language* (EOL)[13] and *Epsilon Transformation Language* (ETL)[14] [98]. EOL is a domain-specific language for creating, querying, and modifying EMF models by means of common programming constructs, as well as first-order logic OCL operations. ETL is a domain-specific language for hybrid, rule-based M2M transformations built on top of EOL. It provides common transformation capabilities, as well as the ability to transform many input to many output models by means of both declarative and imperative transformation specifications, allowing for sophisticated transformation logic, as well as abstraction and reuse.

Going back to the user interface example discussed above, assuming that after adding

[13]See `http://www.eclipse.org/epsilon/doc/eol/`.
[14]See `http://www.eclipse.org/epsilon/doc/etl/`.

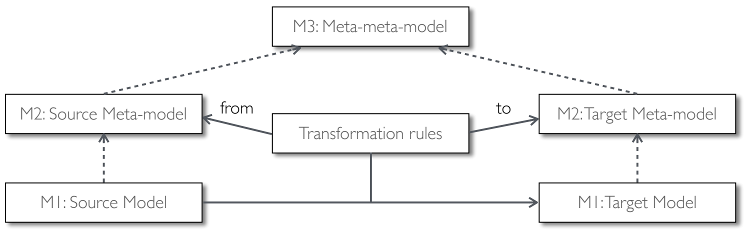
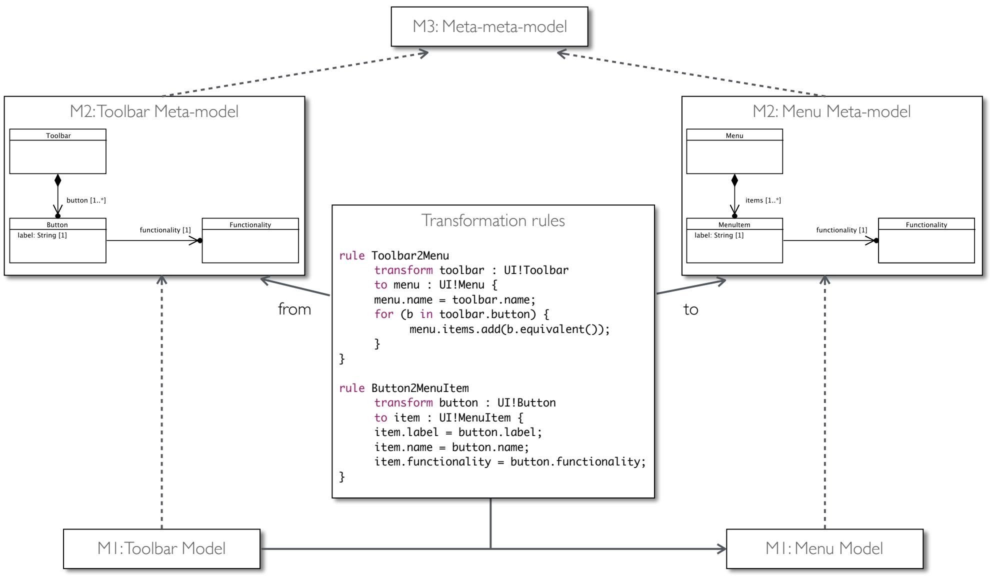


Figure 2.10.: Model to model transformation (based on [13])

the toolbar buttons, there is now a need to add corresponding menu items in a menu, it is necessary to first define corresponding meta-model for the menu which is very similar to the one for the toolbar, containing concepts for *Menu*, which in turn contains *MenuItems* that have a label as well as a *Functionality* than shall be triggered. Figure 2.11 illustrates the transformation specified by means of ETL for this scenario. There are two transformation rules mapping *Toolbar* to *Menu* and *Button* to *MenuItem* correspondingly. During the transformation of the *Toolbar* elements, the *MenuItems* resulting from the transformation of the corresponding *Buttons* are added to the corresponding *Menu* resulting from the transformation of the *Toolbar*. In addition to creating *Menus* automatically from existing *Toolbars*, such transformations can also be used keep *Toolbars* and *Menus* synchronised so that for every new *Button*, corresponding *MenuItem* is created automatically ensuring that the same *Functionality* is accessible through both types of user interfaces. In this scenario, a bi-directional transformation can also ensure that a new *Button* is added for every *MenuItem* added to the *Menu* if both *Toolbars* and *Menus* are modified manually independently from one another and need to be kept consistent.

In this simplified example, the mapping is fairly straightforward one-to-one. In practice this is not necessarily always the case, often multiple concepts from the source model need to be aggregated into one concept from the target model, or one concept from the source model needs to be distributed among several concepts from the target model.

As noted above, M2M transformations can also be performed on the same model. Considering the toolbar example and assuming the functionalities are defined first, a transformation rule can be specified that adds a new button for each functionality automatically.



```
rule Toolbar2Menu
      transform toolbar : UI!Toolbar
      to menu : UI!Menu {
      menu.name = toolbar.name;
      for (b in toolbar.button) {
            menu.items.add(b.equivalent());
      }
}

rule Button2MenuItem
      transform button : UI!Button
      to item : UI!MenuItem {
      item.label = button.label;
      item.name = button.name;
      item.functionality = button.functionality;
}
```

Figure 2.11.: Transformation from toolbars to menus

3. Finding Causes for Events of Interest

While software repositories provide a wealth of information related to the development and evolution of software projects, most of it is of empirical nature, that is, describing consequences rather than causes. For example, developers typically describe their development and maintenance activities as *fixing* issues and problems, *improving* characteristics, *adding* features and functionality, and *refactoring* code. In contrast, during software assessment, we are often more interested in the potential causes for such activities. For example, if a problem in an artifact was fixed during at some point in time leaving the artifact in a fixed state, it is often the case that the same artifact was left in a "broken" state at an earlier point in time. Thus, there was activity which left the artifact in a state that needed subsequent fixing. Such kind of information, although highly valuable, is rarely available in software repositories. This is due to the fact that it is either not known at a given point in time, with developers unknowingly introducing potential faults in their daily development and maintenance activities, despite their best intentions, or, if it is known, based on documented issues that have been addressed, it is very time consuming to add in retrospect.

Within the context of this thesis, we are concerned with the assessment of developer behaviour with respect to certain activities which contribute to causing undesirable phenomena, such as reported failures that need fixing, or difficult to maintain code that needs refactoring and restructuring. We refer to these phenomena as *events of interest*. In this chapter, we explore means for the retrospective identification of the potential causes for events of interest based on empirical data. The potential causes for events of interest can be ultimately designated as *technical risks* based either on individual indicators or a combination of indicators. The generic approach is also suitable for assessment tasks that are not strictly concerned with technical risks. This chapter is based on an extended and revised version of [114].

3.1. Line Tracking

VCSs store information about the evolution of a software project and its artifacts in terms of revisions of files. For textual files, VCSs also store differences between subsequent revisions of a file, referred to as *diffs*. These differences are described in terms of changes to lines, which are typically grouped together in fragments of contiguous changed lines, often referred to as *hunks*. The kinds of changes to fragments include:

- *additions* — new lines are inserted in the subsequent revision, introducing a new fragment,
- *removals* — lines are removed in the subsequent revision, thus removing a fragment, and
- *modifications* — lines are modified in the subsequent revision, where a modified line is typically represented as a line that has been removed and then added; any number of lines may be added and/or removed in addition to the modified lines, thus the number of lines in the subsequent revision may be substantially different from the number of lines in the previous revision of a modified fragment.

This kind of information about the evolution of software artifacts provides the foundation for a number of applications, such as:

- measuring the amount and density of changes by means of *code churn* metrics [90, 91, 115, 127],
- identifying hotspots of frequently changed pieces of code by means of *hot-spot analysis* [102, 115],
- determining the origin of a piece of code by means of *origin analysis* [58, 59], and
- tracking the evolution of a piece of code across revisions by means of *line tracking* [22, 97, 117].

To illustrate the relevant notions, consider the line change map shown in Figure 3.1. A *line change map* [115] is a visual representation of the changes to the lines of a file across revisions as reflected by the *diffs* stored in the VCS. The lines of a file are plotted on the vertical axis, indicating the corresponding line numbers, against the subsequent revisions of the file plotted on the horizontal axis. The contents of the file in the different revisions are overlaid in the respective segments for illustrative purposes. In this artificial example, we are contemplating a file initially containing 10 lines in *revision 1*, to which three lines are added in *revision 2*, followed by modifications to *lines 4–5* and *8–9* in *revision 3*. Further on, in *revision 4* two new lines (*1a–2a*) are added between the original *lines 6* and *7* and *lines 8–9* have been modified again and shifted by two lines due to the addition of the new *lines 1a–2a*. Finally, in *revision 5 lines 4–5* and *7–8* (newly added in *revision 4*) are modified again. The coloured areas represent changed fragments between subsequent revisions, with incremental numbers overlaid on each changed fragment for referencing purposes.

We can then utilise this kind of information to determine the potential causes for events of interest. This is achieved by applying a line-tracking algorithm, such as the ones described in [23, 97], in order to identify when a fragment was last changed (which is also a functionality that is often provided by VCSs) and also in order to track a code fragment across multiple revisions all the way back to its origin.

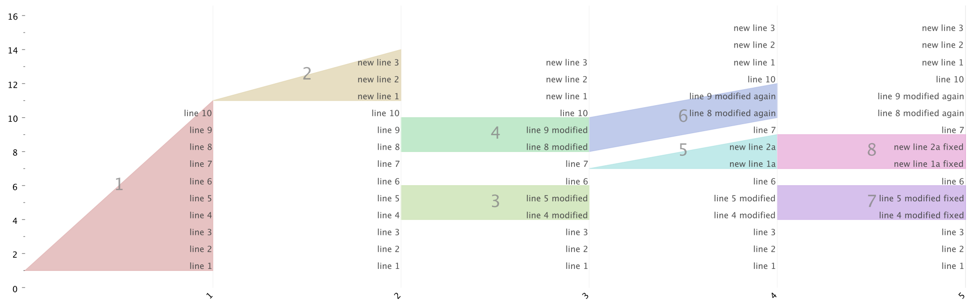


Figure 3.1.: Line change map

Consider for example Figure 3.2, in which the line-tracking for *fragment 3* from Figure 3.1 is visualised by means of red lines tracking the location of the fragment across revisions. Here, it is determined that the content of *fragment 3* was last changed in *revision 1*. Thus, we state tentatively that *revision 1* is the *cause* for the changes to *fragment 3* in *revision 3*. Similarly, applying this approach to the other fragment in *revision 3* (*fragment 4*), we determine that its content was also last changed in *revision 1*. Given this information, we state that *revision 1* is the *cause* for the changes in *revision 3* in this example. Inversely, *revision 3* is referred to as a *fix* (also *effect*) for the changes in *revision 1* in this case. At this point, the notions of *causes* and *fixes* are independent from the nature of the *fix* — it may be a fix for a reported issue or an event of interest in general, but it may also be a change not related to any issue. Note that while the set of revisions identified as *causes* for a given revision is definitive, meaning that no additional causes may be added for that revision, the set of revisions identified as *fixes* for a given revision reflects the state of knowledge at a given point in time, meaning that future revisions may also *fix* issues introduced in that revision.

If we consider *revision 5*, the line-tracking for *fragment 7* as shown in Figure 3.3 indicates that its content was last changed in *revision 3* as *fragment 3*. In contrast, the content of *fragment 8* in *revision 5* was introduced and last changed in *revision 4* as *fragment 5*. In this case, we can then state that both *revision 3* and *revision 4* are determined to be the *causes* for the changes in *revision 5*. Inversely, *revision 5* is referred to as a *fix* for the changes in both *revision 3* and *revision 4* in this case.

3.2. Causes and Fixes

In Section 3.1, we exemplified and outlined relationships between states of an artifact identified as causing and fixing states based on the line-tracking information. In the beginning of this chapter we also outlined the notion of an event of interest. Before we proceed, we need to establish these and other related notions in order to be able to

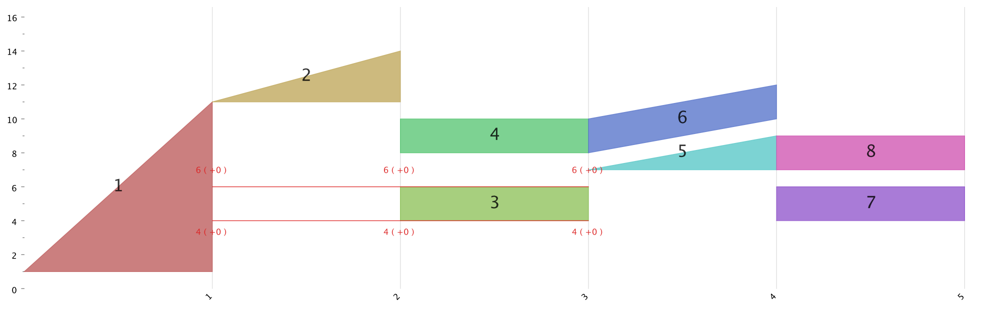


Figure 3.2.: Line tracking for fragment 3

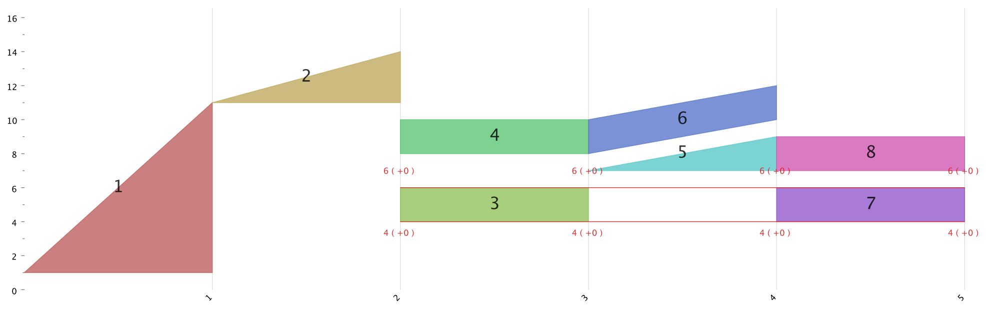


Figure 3.3.: Line tracking for fragment 7

reason about them in formal terms:

Artifact: An entity A at any level of granularity, such as project, file, class, or method, on which developers perform development and maintenance activities. An artifact may contain other artifacts at finer levels of granularity.

State: A revision R_t of artifact A at a point in time t, where $R_t \in REVISIONS$ denoting all the revisions of the artifact.

Event of interest: A state R_t of an artifact A at a point in time t which can be described by some quantitative or qualitative characteristic *factor*, such as the content of a descriptive message associated with the state.

Fix: A modification to an existing part of an artifact in a given state F_t, that was last modified or created in a previous state C_{t-n}, where where $F_t, C_{t-n} \in REVISIONS$. The modification may, but does not strictly need to, relate to fixing a problem.

Cause: A modification of a part of an artifact at a given state C_t that was modified in a later state F_{t+n}, where $C_t, F_{t+n} \in REVISIONS$.

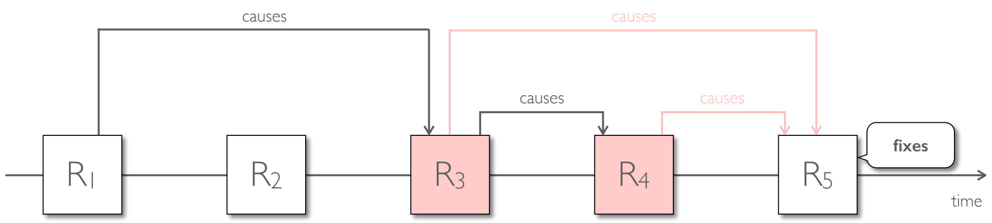


Figure 3.4.: Cause-Fix Graph for Figure 3.1

Cause-Fix Relationship: A relationship $C_{t-n} \xrightarrow{causes} F_t$ between two states (C_{t-n}, F_t) of an artifact A, where a part of A that was modified in C_{t-n} was subsequently modified in a later state F_t, hence C_{t-n} is considered a cause for F_t.

Cause-Fix Graph: A hierarchical directed graph $G_A = (N, E)$, where the set of nodes N includes representations for each state of an artifact A. A state may contain other states at finer levels of granularity based on the containment relationships between the corresponding artifacts for the states. For example, the state for a class may contain also states for methods modified at the same time as the class. The set of directed edges E includes representations for each cause-fix relationship between states of artifact A.

Based on the cause-fix relationships, for a given state F_t identified as a fix, we define the set of states fixed by F_t (i.e. the set of causes for F_t) as:

$$F_t^{\text{FIXES}} = \{C_{t-n} \in REVISIONS | C_{t-n} \xrightarrow{causes} F_t\} \tag{3.2.1}$$

Conversely, for a given state C_{t-n} identified as a cause, the set of known caused fixes for C_{t-n} is defined as:

$$C_{t-n}^{\text{CAUSES}} = \{F_t \in REVISIONS | C_{t-n} \xrightarrow{causes} F_t\} \tag{3.2.2}$$

A cause-fix graph for each artifact can be constructed by utilising information extracted from VCSs and applying the line tracking approach described in Section 3.1 or any of the approaches for tracking the location of modified fragments already described in the literature [23, 187]. A visualisation of such a graph for the example from Figures 3.1–3.3 is shown in Figure 3.4, where the cause-fix relationships for the states at the project level are shown.

The cause-fix relationships help us identify the potential causes for any event, but we are primarily concerned with finding causes for events of interest. For illustrative purposes we will consider the changes in *revision 5* to be one such event of interest. In this case, identifying *revision 5* as an event of interest is based on the content of the changes themselves — as illustrated in Figure 3.1, *lines 4–5* and *8–9* in *revision 5* are considered to be "fixing" the previous content of these lines. Based on this property of *revision 5* and the constructed cause-fix graph, we can identify *revision 3* and *revision 4* as the potential causes for the event of interest. Both states as well as the relationships between them and the "fixing" state are highlighted in Figure 3.4 by using a different colour, red in this case. While in reality such explicit labeling is only rarely present in the form of comments, there is usually some indication that something was "fixed" within the revision description message in the VCS. Alternatively, a revision can be described as an event of interest based on changes to measurable attributes, such as recorded measurements for size, complexity, and documentation density. In the approach discussed in this chapter, it is assumed that there are such means or that there are labels for events of interest already present in the data obtained from the VCS.

3.3. Weights and Factors

The simplified binary classification of nodes in the graph as causes for events of interest discussed in Section 3.2 presents two fundamental limitations. The basic artificial example from Figure 3.4 already illustrates these, raising the following questions related to the significance of the classification:

1. Given that both *revision 3* and *revision 4* are identified as causes for the fixes in *revision 5*, are they both equally likely causes and thus to be considered of equal importance?

2. Given that *revision 3* is identified as causing both *revision 4* and *revision 5*, is it then considered a less likely cause for *revision 5*, and thus to be considered of less importance?

In order to reason about these questions, we need means to quantify the relationships between *fixes* and *causes*. Even from the rather simple example discussed so far, we can already establish that *cause-fix* relationships are *many-to-many*, that is, a cause may lead to many subsequent fixes, and fix may address multiple previous causes. Conceptually, we consider a fix as an activity that is "removing a weight" from a state of an artifact. Consequently, the activities that contributed to the causes for the fix "added weight" to the corresponding states of the artifact. Our approach for the quantification of the degree to which a revision can be considered as the cause for another revision is based on this conceptual premise. In addition, there may be different types of "weights" based on different characteristics of the fixing revision, e.g. "fixing an issue", "refactoring code", etc., reflecting the different kinds of events of interest. In order to accommodate multiple independent types of weights, we extend and generalise the notion to "removing a weight related to a weight factor *wf*", where $wf \in \{\text{fixes}, \text{refactors}, \ldots\}$. Thus, we speak of a fixing state F_t as having *removed weight* (rw) with respect to weight factor wf, where:

$$rw(F_t, wf) = \begin{cases} 1 & \text{if } wf \text{ property holds for } F_t \\ 0 & \text{otherwise} \end{cases} \tag{3.3.1}$$

Conversely, each of the causes can be regarded as contributing to that weight, thus for each cause-fix relationship $C_{t-n} \xrightarrow{causes} F_t$ and for each weight factor wf, we define the notion of *contributed weight* (cw) of a causing revision C_{t-n} to a fixing revision F_t with regard to a weight factor wf as:

$$cw(C_{t-n}, F_t, wf) = \frac{rw(F_t, wf)}{|F_t^{\text{FIXES}}|} \tag{3.3.2}$$

For each fix F_t caused by a causing state C_{t-n}, the causing state C_{t-n} is then said to accumulate a *total weight* (tw) with regard to weight factor wf, defined as:

$$tw(C_{t-n}, wf) = \sum_{F_t \in C_{t-n}^{\text{CAUSES}}} cw(C_{t-n}, F_t, wf) \tag{3.3.3}$$

For example, if the fix in *revision 5* (R_5) is removing a weight $rw(R_5, \text{fixes}) = 1$ with respect to the weight factor "fixes", and if there are two revisions $R_5^{FIXES} = \{C_{5-n} \in REVISIONS | C_{5-n} \xrightarrow{causes} R_5\} = \{R_3, R_4\}$ identified as causes for this fix, that are considered to be contributing equally to that weight, then each cause-fix relationship is contributing a weight $cw(C_{5-n}, R_5, \text{fixes}) = 0.5$. On the other hand, since R_4 can be considered neutral with respect to the "fixes" weight factor (i.e. $rw(R_4, \text{fixes}) = 0$), as it is not identified as an event of interest, hence R_3 does not contribute any weight to R_4 (i.e. $cw(R_3, R_4, \text{fixes}) = 0$). In this case, we speak of R_3 and R_4 as having a $tw(R_3, \text{fixes}) = tw(R_4, \text{fixes}) = 0.5$. Thus, at first glance it may seem that R_3 and R_4 can be considered equally important.

In order to reason about the second question, we need to contemplate the inverse relationship. If we consider R_3 in the example, it causes both R_4 and R_5, thus $R_3^{CAUSES} = \{F_{3+n} \in REVISIONS | R_3 \xrightarrow{causes} F_{3+n}\} = \{R_4, R_5\}$, whereas R_4 only causes R_5, i.e. $R_4^{CAUSES} = \{R_5\}$. To take this into account, we define the notion of *average weight* (aw) with regard to weight factor wf as:

$$aw(C_{t-n}, wf) = \frac{tw(C_{t-n}, wf)}{|C_{t-n}^{\text{CAUSES}}|} \tag{3.3.4}$$

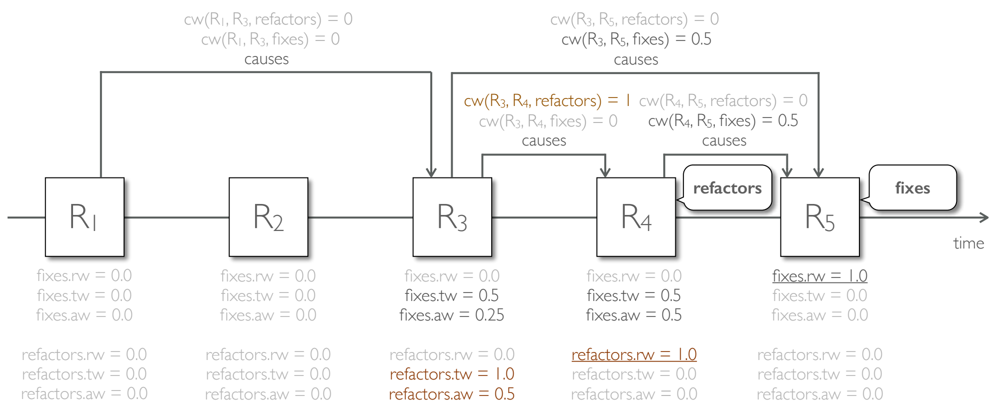


Figure 3.5.: Cause-Fix Graph for Figure 3.1 (with details)

In the example above, this yields $aw(R_3, \text{fixes}) = 0.25$ and $aw(R_4, \text{fixes}) = 0.5$, respectively. Thus, we can state that while both R_3 and R_4 can be considered important as causes for the fix in R_5 with respect to the weight factor “fixes”, since R_3 is also a cause for R_4, it is less important than R_4 as it also caused a “neutral” change with respect to the factor “fixes” in addition to the fixing change.

If we consider a different weight factor — let’s assume that R_4 is identified as an event of interest of a different kind, e.g. a “refactoring", the removed weight with respect to the “refactoring" weight factor for R_4 is then $rw(R_5, \text{refactors}) = 1$. Consequently, the weights for the “refactors” weight factor are distributed differently, where R_3 is the only identified cause contributing all the removed weight, hence $cw(R_3, R_4, \text{refactors}) = tw(R_3, \text{refactors}) = aw(R_3, \text{refactors}) = 1$. The detailed view of the cause-fix graph from Figure 3.4 shown in Figure 3.5 includes the corresponding removed weight, total weight, and average weight for each weight factor in every state. In addition, it also includes annotations for the contributed weights for each weight factor on the cause-fix relationships between the states.

Similarly, arbitrary other factors can be considered in order to accommodate different assessment tasks, focusing on different characteristics or even combinations of characteristics of states that can serve as indicators for events of interest. In addition, a “default” weight factor with $rw(F_t, \text{default}) = 1$ for all revisions can serve as a baseline for all other weights.

Note again, that while information about the causing states for a given fix can be considered definitive, information about the fixing states for a given state is only partially known as far as the data indicates up to a given point in time. Future states may still include fixes for already existing states, thus also potentially altering their weights.

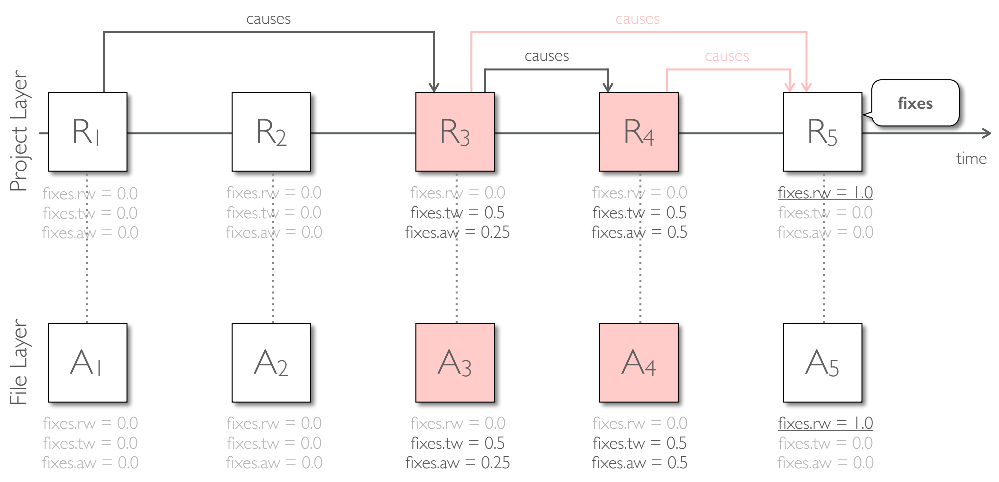


Figure 3.6.: Layered Cause-Fix Graph for Figure 3.1

3.4. Layers and Granularities

In the examples considered so far, only the project level of granularity was considered for simplicity, assuming that only one file was modified in each revision of the project. In this case, the assigned weights will also be identical at both project and file levels of granularity, thus we can simply copy the weights from the project states to the file states, as shown in Figure 3.6. In practice, this simplification is rarely applicable as usually multiple (and often related) files are changed together as part of a revision, thus a state at the project level would contain multiple states at the file level. Furthermore, once additional levels of granularity are considered, such as the logical level of e.g. methods, classes, modules, and functions, even changes within the same file often affect multiple logical entities within the file. This may result in multiple states at the logical level contained in a single state at the file level.

A further complication stems from the fact that while a set of related artifacts may be changed within a fixing revision, only a subset of these artifacts and possibly a set of additional artifacts may be changed within a corresponding causing revision. Thus, the causes and fixes for a state of an artifact at a finer level of granularity may be a subset of the causes and fixes for a containing state at a coarser level of granularity. Consequently, the weight distribution may also vary across the different levels of granularity. Two fundamental challenges emerge as a result:

1. Given a state that is identified as the cause for a fix, where the state contains multiple states of artifacts at a finer level of granularity, are all of the states at the finer level of granularity contributing equally to the cause for the fix?

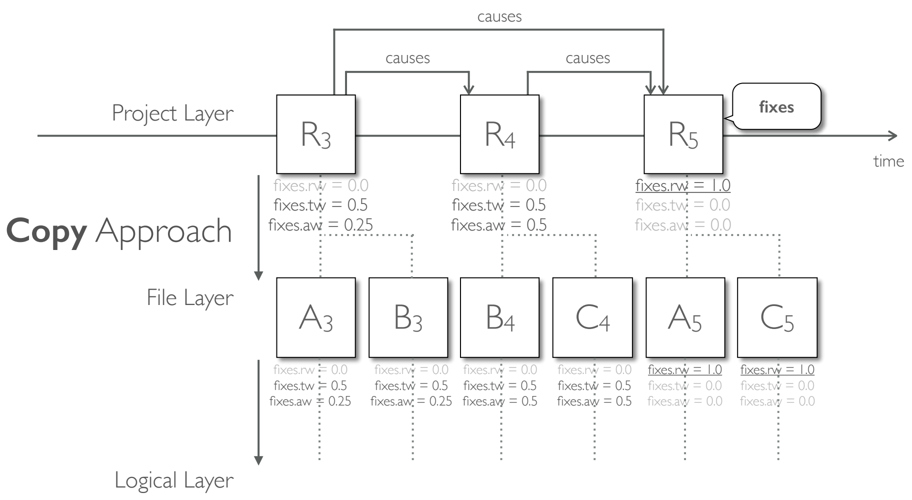


Figure 3.7.: Copy approach for distributing weights across different levels of granularity

2. Given a state that is identified as a fix, where the state contains multiple states of artifacts at a finer level of granularity, are all of these states at finer level of granularity contributing equally to the fix?

To illustrate the first challenge, consider a different scenario, sketched in Figure 3.7. In this scenario, there are three files, A, B, and C, two of which are modified as part of *revisions 3*, *4*, and *5*. There are two states at the file level for each state at the project level. The naive approach would be to simply copy the weights from the project level to the file level. With regard to the first challenge, the question arises whether the states B_3 and B_4 at the file level are contributing at all to the cause for the fix in R_5, given that in R_5 only A and C have been modified. In other words, shall B_3 and B_4 be assigned any weights at all? The same is also applicable at the logical level.

Even from this simplified example, we can observe that the naive copy approach can potentially result in a lot of noise since the sets of states of artifacts at a finer level of granularity may vary between the causing and the fixing states at the coarser level of granularity. A more adequate approach is to construct a distinct cause-fix graph at each layer corresponding to a given level of granularity based on the cause-fix relationships among the states at that level. This enables weight redistribution within the corresponding layers, yielding more accurate weighting for each layer. Consider the same scenario from Figure 3.7, where instead of copying the weights from the project layer, we calculate the weights at the file layer based only on the cause-fix relationships at that layer, as illustrated in Figure 3.8. This approach yields more accurate weight distribution, taking into account that only A and C were modified as part of the fix in R_5. Hence, the corresponding states A_3 and C_4 carry the full responsibility for causing the fix in R_5 and

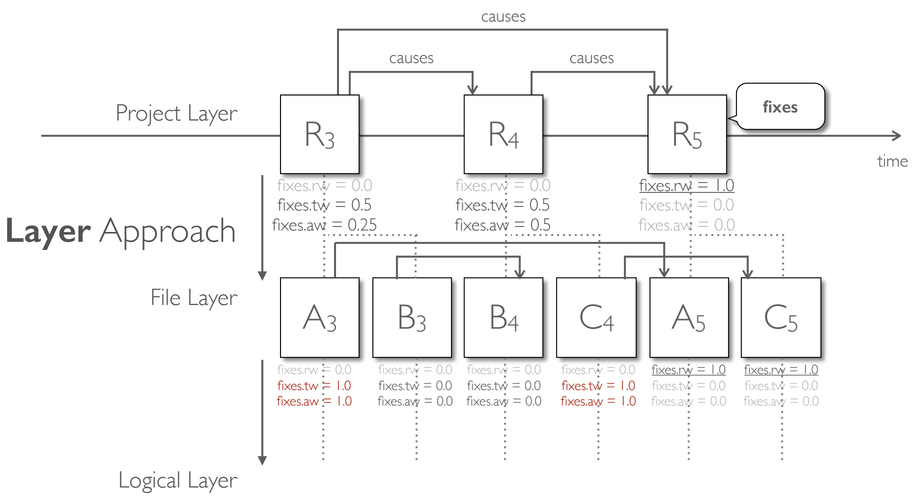


Figure 3.8.: Layer approach for distributing weights across different levels of granularity

thus shall be assigned the corresponding weights, whereas the states B_3 and B_4 can be considered neutral in this case and shall be assigned no weights at all.

This brings us to the second challenge, which can be exemplified in the given scenario as follows: given that both states A_5 and C_5 at the file level are considered as part of the fix in R_5 at the project level, are both A_5 and C_5 contributing equally to the fix in R_5? So far, states at finer levels of granularity simply inherited the removed weights from the containing state at a coarser level of granularity, that is $rw(A_5, \text{fixes}) = rw(C_5, \text{fixes}) = rw(R_5, \text{fixes})$. Inheriting the removed weights from the containing state does not take into account potential dilution of the contribution of each individual state at the finer level of granularity. If there is a single state at the finer level of granularity, it can be considered solely responsible for the fix, but if there are a large number of states at the finer level of granularity, each one of them may be contributing only a small part to the fix.

Even in this simple artificial scenario, we need to account for both the number of states at a finer level of granularity involved in a fix and potentially also other characteristics of each state in order to obtain a more accurate picture. This raises the following concerns that need to be taken into account:

- Does the number of states of artifacts at a finer level of granularity involved in a fix dilute the contribution of each individual state to the fix?
- Do states of certain types of artifacts contribute more to a fix than others (e.g. states of code vs. image artifacts)?
- Do states of larger artifacts contribute more to a fix than states of smaller artifacts?

- Do states of artifacts containing larger changes contribute more to a fix than states of artifacts containing smaller changes?

In order to take these concerns into account in the weighting approach, we define different weight distribution strategies, which distribute removed weights in fixing states across artifact states at finer levels of granularity depending on their contribution to a fix. Consequently, the weights calculated for the causing states are also updated according to the strategy being used.

3.5. Weight Distribution Strategies

As noted in Section 3.4, when we consider the contribution of each state of an artifact at a finer level of granularity to a fix in a state of a containing artifact at a coarser level of granularity, we need to take different aspects into account. These include as the number of states at the finer level of granularity involved in the fix, the type and size of the corresponding artifacts, as well as the amount of change to each corresponding artifact. To address these concerns, we define four weight distribution strategies which refine the notion of *removed weight* (*rw*) to *distributed removed weight* (*drw*). The distributed removed weight according to a *distribution strategy ds* for a state of artifact A_t contained in a state R_t is defined based on the following expression:

$$drw(A_t, wf, ds) = rw(R_t, wf) \cdot df(A_t, ds) \tag{3.5.1}$$

where the *distribution factor* (*df*) for a distribution strategy *ds*, denoted as $df(A_t, ds)$, determines the proportion of the removed weight from the containing state R_t allocated to the contained state A_t according to the distribution strategy of choice. As a baseline, the distribution factor for the *inherit* strategy discussed in Section 3.4 and shown in Figure 3.8 can be defined as:

$$df(A_t, \text{inherit}) = 1 \tag{3.5.2}$$

Substituting the removed weight with the distributed removed weight in the calculation of the contributed weights enables the support for distributed removed weights according to a given strategy throughout the approach.

3.5.1. Shared Strategy

The *shared* strategy takes into account number of states of artifacts at a finer level of granularity involved in a fix based on the assumption that a large number of states dilutes

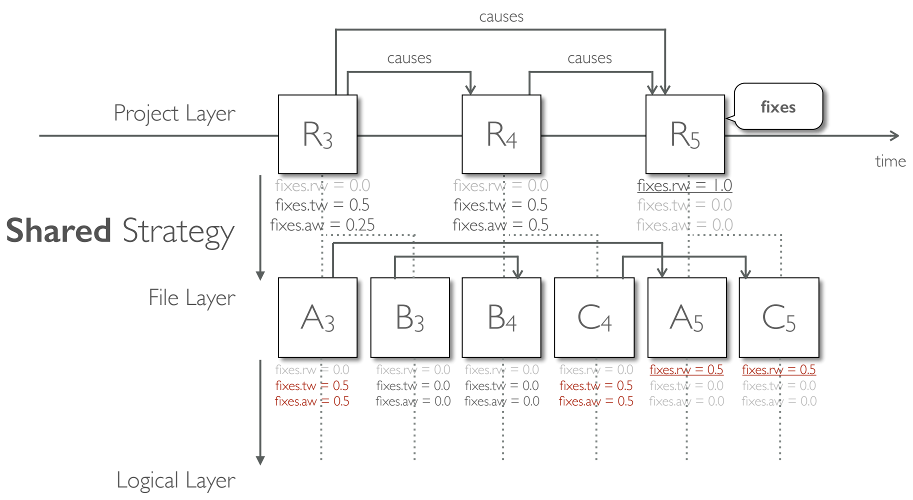


Figure 3.9.: Shared strategy for distributing removed weights across layers

the contribution of each individual state to the fix. This strategy distributes the removed weight equally, assuming that each state at a finer granularity contributes equally to the fix. As a consequence, the more states contributing to a fix the less impact each individual state has. The distribution factor for the *shared* strategy is defined as:

$$df(A_t, \text{shared}) = \frac{1}{|R_t^{CONTENTS}|} \tag{3.5.3}$$

where $R_t^{CONTENTS}$ denotes the set of states at a finer level of granularity contained in state R_t and $A_t \in R_t^{CONTENTS}$.

The application of the *shared* strategy to the running example from Figures 3.7–3.8 and the resulting weight redistribution is shown in Figure 3.9. Since two states at the file level of granularity are involved in the fix at the project level of granularity, the $df(A_5, \text{shared}) = df(C_5, \text{shared}) = 0.5$ and hence $drw(A_5, \text{fixes}, \text{shared}) = drw(C_5, \text{fixes}, \text{shared}) = 0.5$. Consequently, the total and average weights of the corresponding causing states at the file level of granularity are also adjusted. Thus, the dilution of the contribution of each state at the finer level of granularity to the fix is also extended to the total and average weights of the corresponding causing states.

While we exemplify only the application of the strategy to the project and file levels of granularity, this strategy is also applicable at different logical levels of granularity. Note, however, that it shall be applied at each logical level of granularity (e.g. Class, Method, Function, etc.) separately, which makes its application at that level more similar to the *type* strategy.

3.5.2. Type Strategy

The *type* strategy takes into account how much states of artifacts at a finer level of granularity contribute to a fix based on the *type* (*at*) of the corresponding artifact. This strategy distributes the removed weight equally among states of artifacts of a selected type (indicated as a parameter), while states of artifacts of other types do not get any removed weight assigned. It can be used to emphasise the importance of states of code artifacts and de-emphasise the importance of image artifacts, for example. The distribution factor for the *type* strategy for a given type T is defined as:

$$df(A_t, \text{type}(T)) = \begin{cases} \frac{1}{|\{s_t \in R_t^{CONTENTS} : at(s_t) = T\}|} & \text{if } at(A_t) = T \\ 0 & \text{otherwise} \end{cases} \tag{3.5.4}$$

where $\{s_t \in R_t^{CONTENTS} : at(s_t) = T\}$ denotes the set of states of artifacts of type T contained in R_t.

The application of the *type* strategy for the type *code* to the running example from Figures 3.7–3.9 and the resulting weight redistribution is shown in Figure 3.10. Of the two states at the file level of granularity involved in the fix at the project level of granularity, only A_5 is of type *code*, hence $df(A_5, \text{type(code)}) = 1$, whereas $df(C_5, \text{type(code)}) = 0$ since $at(C_5) = image$. As a result $drw(A_5, \text{fixes}, \text{type(code)}) = 1$ and $drw(C_5, \text{fixes}, \text{type(code)}) = 0$. The total and average weights of the corresponding causing states at the file level of granularity are adjusted respectively. Thus, the emphasis on the contribution of states of code artifacts to the fix is also extended to the total and average weights of the corresponding causing states.

This strategy can be applied multiple times for different types of artifacts, essentially resulting in a distribution of removed weights "within type", i.e. the removed weight of a fixing state at the project level of granularity is distributed once among all states of code artifacts, then again independently among all states of test artifacts, and so on. In a similar manner, it can also be applied at the different logical levels of granularity (e.g. Class, Method, Function, etc.) individually in order to obtain the equivalent of the *shared* strategy at the file level of granularity applied at the logical levels of granularity.

3.5.3. Size Strategy

The *size* strategy emphasises the impact of the size of an artifact (*as*) in a given state that is considered as a part of a fixing state at a coarser level of granularity. The underlying assumption is that larger artifacts require more time and effort to maintain [4] and thus potentially contribute more to the occurrence of an event of interest, such as a fix. Hence, if there is weight to be removed in a fix, the chunk of that weight to be removed from a given artifact is assumed to be proportional to the size of the artifact. The size

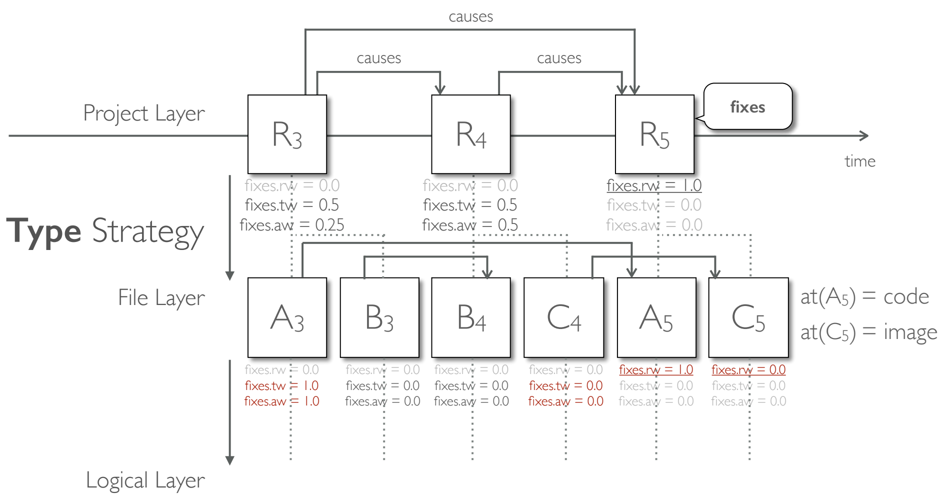


Figure 3.10.: Type strategy for distributing removed weights across layers

of an artifact is generally measured in terms of *Lines of Code* (LOC), however other measures may be used as well. The distribution factor for the *size* strategy is defined as:

$$df(A_t, \text{size}) = \frac{as(A_t)}{cs(R_t)} \tag{3.5.5}$$

where the *content size* (cs) for a state of an artifact R_t is the sum of the sizes of all artifacts in the states contained in R_t, defined as:

$$cs(R_t) = \sum_{s_t \in R_t^{CONTENTS}} as(s_t) \tag{3.5.6}$$

The application of the *size* strategy to the running example from Figures 3.7–3.10 and the resulting weight redistribution is shown in Figure 3.11. Given the artifact sizes $as(A_5) = 40$ and $as(C_5) = 60$, the corresponding distribution factors are $df(A_5, \text{size}) = 0.4$ and $df(C_5, \text{size}) = 0.6$, which are also identical to the respective distributed removed weights for A_5 and C_5. The total and average weights of the corresponding causing states at the file level of granularity are also adjusted respectively, emphasising the impact of the size of the corresponding artifacts in the fixing state on their contribution to the fix as indicated by the removed weight assigned to them, and also on the total and average weights of the corresponding causing states.

Similar to the *shared* strategy, the *size* strategy shall be applied at each logical levels of granularity (e.g. Class, Method, Function, etc.) separately, which effectively results

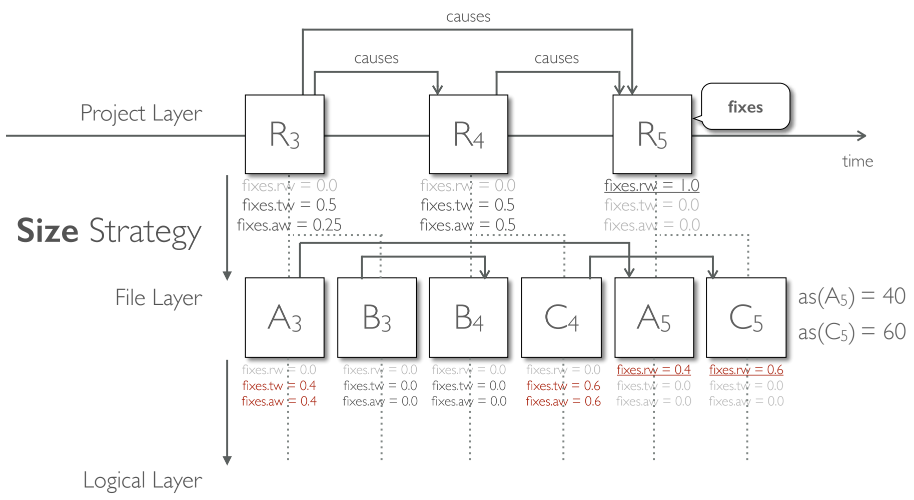


Figure 3.11.: Size strategy for distributing removed weights across layers

in a refinement of the *size* strategy that also integrates the *type* strategy. In that case, the *size* strategy takes a parameter T denoting the type of artifacts it shall be applied to. This refinement is integrated in the distribution factor as:

$$df(A_t, \text{size}(T)) = \begin{cases} \frac{as(A_t)}{tcs(R_t,T)} & \text{if } at(A_t) = T \\ 0 & \text{otherwise} \end{cases} \tag{3.5.7}$$

where the *typed content size* (tcs) for a state of an artifact R_t and an artifact type T is the sum of the sizes of all artifacts of type T in the states contained in R_t, defined as:

$$tcs(R_t, T) = \sum_{s_t \in \{c_t \in R_t^{CONTENTS} : at(c_t) = T\}} as(s_t) \tag{3.5.8}$$

Apart from the application at the logical levels of granularity, this refinement also combines the emphasis on the type and the size of the artifact. When applied at the file level of granularity, only the size of artifacts of the given type is taken into consideration. If a fixing state at the project level includes states of artifacts of different types, e.g. *code* and *test*, and we are interested primarily in artifacts of type *code*, the typed *size* strategy distributes the removed weight according to the size of code artifacts only. Thus, even if the fixing state contains large test artifacts, they will have no impact on the weight distribution among the code artifacts.

3.5.4. Churn Strategy

The *churn* strategy emphasises the impact of the amount of change (churn) of an artifact (*ac*) in a given state that is considered as a part of a fixing state at a coarser level of granularity. The underlying assumption is that larger changes in artifacts require more time and effort to perform and potentially contribute more to the occurrence of an event of interest, such as a fix. Hence, if there is weight to be removed in a fix, the chunk of that weight to be removed from a given artifact is assumed to be proportional to the amount of change that needed to be performed in the artifact. The distribution factor for the *churn* strategy is defined as:

$$df(A_t, \text{churn}) = \frac{ac(A_t)}{cc(R_t)} \tag{3.5.9}$$

where the *content churn* (*cc*) for a state of an artifact R_t is the sum of the churn for all artifacts in the states contained in R_t, defined as:

$$cc(R_t) = \sum_{s_t \in R_t^{CONTENTS}} ac(s_t) \tag{3.5.10}$$

The application of the *churn* strategy to the running example from Figures 3.7–3.11 and the resulting weight redistribution is shown in Figure 3.12. Given that $ac(A_5) = 4$ and $ac(C_5) = 1$, the corresponding distribution factors are $df(A_5, \text{churn}) = 0.8$ and $df(C_5, \text{churn}) = 0.2$, which are also identical to the respective distributed removed weights for A_5 and C_5. The total and average weights of the corresponding causing states at the file level of granularity are also adjusted respectively. This emphasises the impact of the amount of change in the states of the corresponding artifacts in the fixing state on their contribution to the fix. Their contribution is indicated by the removed weight assigned to them. By extension, this also emphasises the impact of the amount of change on the total and average weights of the corresponding causing states.

Contemplating the application of both the *size* and the *churn* strategies, as illustrated in Figure 3.11 and Figure 3.12, respectively, we may observe a contradiction in the weight distributions. The *size* strategy indicates that C_5 is contributing more to the fix in R_5 due to its larger size and hence its causing state C_4 is the more likely cause for the fix in R_5. On the other hand, the *churn* strategy indicates that A_5 is contributing more to the fix in R_5 due to the larger amount of change in A_5 and hence its causing state A_3 is the more likely cause for the fix in R_5. The different strategies ultimately enable emphasising different characteristics of events of interest. Which one is to be used depends on the application context and the assessment task. If the size of artifacts is perceived as resulting in more effort involved in understanding them during main-

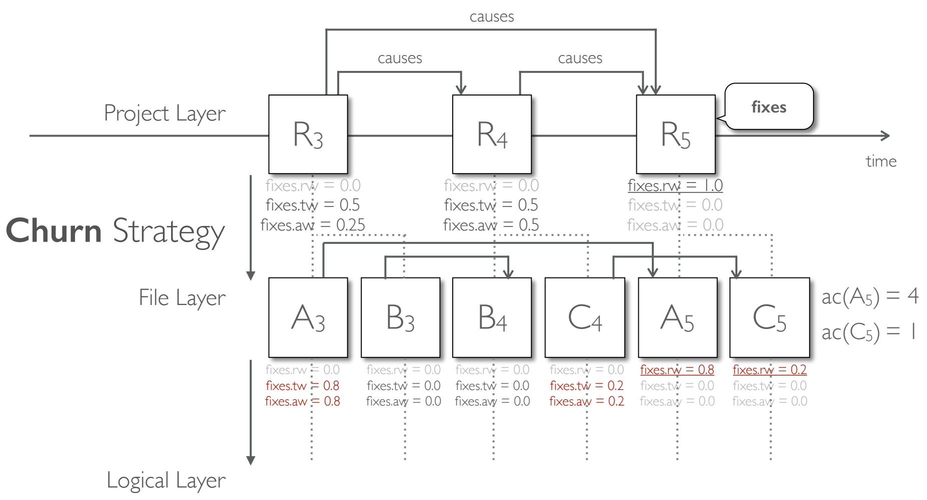


Figure 3.12.: Churn strategy for distributing removed weights across layers

tenance and development tasks, then the *size* strategy will be more adequate. On the other hand, if the amount of change in states of artifacts is considered more critical with respect to the effort involved in performing maintenance and development tasks, then the *churn* strategy will be more adequate. The states of artifacts that contribute both to events of interest and to their likely causes can be identified and emphasised based on their relative importance with respect to the effort involved in modifying them.

There are different kinds of churn measures described in the literature [90, 91, 115, 127]. We consider a rather simple absolute measure of churn defined as the sum of additions and removals in terms of lines (*Churned LOC* in [127]), where a modification is considered both a removal and an addition of one or more lines that are part of the modification. Other notions of churn can also be used in the *churn* strategy, however if a relative churn measure is used, such as the ones described in [127], the distribution factor may need to be adjusted as well.

Similar to the *shared* and the *size* strategy, the *churn* shall be applied at each logical level of granularity (e.g. Class, Method, Function, etc.) separately. In that case, the *churn* strategy takes a parameter T denoting the type of artifacts it shall be applied to. This refinement is integrated in the distribution factor as:

$$df(A_t, \text{churn}(T)) = \begin{cases} \frac{as(A_t)}{tcc(R_t,T)} & \text{if } ac(A_t) = T \\ 0 & \text{otherwise} \end{cases} \tag{3.5.11}$$

where the *typed content churn* (tcc) for a state of an artifact R_t and an artifact type T is the sum of the sizes of all artifacts of type T in the states contained in R_t, defined as:

$$tcc(R_t, T) = \sum_{s_t \in \{c_t \in R_t^{CONTENTS} : at(c_t) = T\}} ac(s_t) \tag{3.5.12}$$

Apart from the application at the logical levels of granularity, this refinement also combines the emphasis on the type of the artifact and the amount of change in the artifact. When applied at the file level of granularity, only the churn of artifacts of the given type is taken into consideration. If a fixing state at the project level includes states of artifacts of different types, e.g. *code* and *test*, and we are interested primarily in artifacts of type *code*, the typed *churn* strategy distributes the removed weight according to the churn of code artifacts only. Thus, even if the fixing state contains large changes to test artifacts, they will have no impact on the weight distribution among the code artifacts. Similar to the *type* and the typed *size* strategy, the typed *churn* strategy can be applied multiple times for different types of artifacts, essentially resulting in a distribution of removed weights "within type".

3.6. Related Work

Existing approaches are typically based on some form of origin analysis [58], involving line tracking and annotation graphs [97], line histories [22], line mapping [117], as well as refinements to these [23, 187] in order to map and track entities across revisions. Historage [67] is an approach for tracing fine-grained entity histories including renaming changes. The approach presented in this chapter builds on top of these approaches, applying origin analysis to events of interest in order to determine their potential causes and then quantifying the cause-fix relationships by means of weights. Our approach also considers different levels of granularity. Any of the existing approaches can be used as a foundation and generally the accuracy of the weighting depends in part on the quality of the results from the underlying origin analysis approach.

Different applications for the existing approaches have been discussed in the literature, ranging from finding fix-inducing changes [165] and understanding the role of authorship on implicated code [144] to defect-insertion circumstance analysis [142]. While such applications do serve a similar purpose — identifying potential causes for events of interest, they are focused on identifying causes before the event of interest has occurred. Such applications generally require sufficient information about known causes for events of interest, which serves as training data in order to build pattern recognition models that are then used to identify potential causes for events of interest. Both, the training and the validation of such pattern recognition models requires data annotated with known causes for events of interest. The approach discussed in this chapter can be applied to produce such data emphasising different characteristics across multiple levels of granularity for different kinds of events of interest.

The challenge of "tangled changes" [76] is somewhat related to the topic of this chapter. The authors study the prevalence of changes that are unrelated or loosely related to events of interest and apply a multi-predictor approach to untangle them, based on different confidence voters. The approach discussed in this chapter relies on weighting and different weight distribution strategies to emphasise certain characteristics of changes related to events of interest, that are considered to be of importance in a given context. It can further benefit from a more sophisticated untangling approach such as the one described in [76], which can be incorporated as an additional weight distribution strategy to refine the distribution of weights among fixing and causing states of artifacts across the different levels of granularity.

To the best of our knowledge none of the existing approaches has incorporated quantification of the extent to which a change in one state contributes to a subsequent fix in a later state of an artifact. Also, none of the approaches has explored how to apply cause-fix analysis across multiple levels of granularity.

3.7. Summary

In this chapter, we discussed an approach for finding potential causes for events of interest in software repositories. An event of interest can be any occurrence that may be of relevance for an assessment task, such as fixing issues and problems, improving properties, adding features and functionality, and refactoring code. The approach adds quantitative information on top of existing approaches for origin analysis, such as ones based on line tracking. The quantitative information is in the form of weights, where an event of interest regarded as a fix is considered to be removing a weight, and the potential causes for the event of interest are considered to be contributing to that weight. Distinct weights can be calculated across different dimensions, based on the kind of event of interest, such as a bug fix, refactoring, etc., designated by a factor for each kind of interest. The approach accommodates weight redistribution across multiple layers corresponding to different levels of granularity in order to provide more accurate information at these levels of granularity. We outlined different strategies for weight redistribution across the different levels of granularity, which enable emphasising different characteristics of the states of artifacts involved in an event of interest, such as their type, size, or the amount of change they have undergone. The emphasis on different characteristics allows us to account for the importance of these characteristics in the effort involved in performing an activity that leads to an event of interest or its causes. Further weight distribution strategies may be defined in order to emphasise other characteristics or combinations of characteristics of events of interest.

There are different related approaches described in the literature, which seek to establish relationships between fixes and their likely causes. However, none of them have incorporated quantification of the extent to which a likely cause contributes to a subsequent fix, especially across multiple levels of granularity. The presented approach builds on top of these approaches and generally any of them can serve as a foundation, providing the relationships between fixes and their likely causes. Based on these relationships, the proposed approach can be used to calculate the corresponding weights and quantify the cause-fix relationships. There are also different related applications discussed in the literature which can be used for similar purposes. However, their scope and focus is mostly on identifying potential causes for events of interest, where the event of interest has not yet occurred. The approach discussed in this chapter can be applied to provide necessary information for the configuration, validation, and refinement of such applications.

While the set of revisions identified as causes for a given revision is definitive, meaning that no additional causes may be added for that revision, the set of revisions identified as fixes for a given revision reflects the state of knowledge at a given point in time, meaning that future revisions may also fix issues introduced in that revision. This affects the reliability of the calculated weights. In future work, a suitable cut-off point in time needs to be defined, after which the calculated weights for causing states can be considered unreliable. Such a cut-off point may be based on release tags, or on the distance between causing and fixing states with respect to a particular factor, or on the distance between causing and fixing states in general.

4. Characterising Developer Behaviour

Developer-related information has already been used for software assessment tasks in the literature [57, 163, 184]. Most approaches focus on defect prediction and typically make use of the author of the code, and perhaps some basic indication of their experience, such as the number of activities they have performed, or the amount of time they have been working on the project. This information is typically combined with characteristics related to the artifacts, such as their size and complexity, as well as characteristics related to the process, such as the number of changes within a period of time, and utilised to build project-specific prediction models. However, project-specific modelling ignores the differences in the behaviour of individual developers. Developers have been indicated to exhibit different programming styles [84, 163, 184]. In addition, they usually have different amount of experience with the project as a whole as well as with certain parts of it. Consequently, developers may have different strengths and weaknesses resulting in different defect patterns [51, 84, 163, 184].

In contrast, when contemplating a project as a whole, differences in the behaviour of individual developers are diffused, which can result in noisy assessment results. Frequently there are organisation- and project-specific guidelines and policies seeking to normalise the behaviour of developers. However, as developer exhibit different strengths and weaknesses, feedback based on a global project-specific assessment will likely have different applicability for different developers. As a consequence, the acceptance for such global assessment may suffer due to the lack of specificity.

The remainder of this chapter is structured as follows: Section 4.1 contains a high-level conceptual overview of the approach for characterising developer behaviour. A detailed breakdown of the different characteristics across five dimensions is included in Section 4.2. Means for making sense of the collected data regarding the various characteristics and for identifying potential patterns, as well as applications for gaining further insights are discussed in Section 4.3. Related work is discussed in Section 4.4.

4.1. Conceptual Overview

Software products are comprised of various artifacts, including source code files, as well as logical constructs within source code, such as classes and methods in object-oriented software development. When reasoning about different characteristics of software during software assessment, the first aspect to look at are the characteristics of the artifacts comprising the software product. This can be referred to as *artifact-centric* software assessment. It can be considered as the first kind of software assessment. As a software product evolves over time, the related artifacts and their characteristics evolve as well. In addition, there are further characteristics describing the evolutionary changes themselves. We refer to an assessment considering change-related characteristics as well as changes to artifact characteristics as *change-centric* software assessment. Since changes do not occur by themselves, but are rather the result of activities performed by developers, the natural next step in software assessment is to consider developer-related characteristics in software assessment as well. It is this kind of assessment, which we refer to as *developer-centric* software assessment, that is the central topic of this thesis.

The first challenge is to *determine what constitutes developer behaviour and how it can be characterised.* As noted above, artifacts can be measured in different ways, thus they are described by different measurable characteristics, where the values of these characteristics at a given time point determine the state of the artifact. Activities performed on an artifact determine the observable changes to the values of the measurable characteristics of the artifact between the state on which an activity was performed and the new state emerging as a result of the activity. Traces related to both the states of the artifacts, as well as the activities performed on them can be collected from the different kinds of software related assets, such as VCSs and ITSs. Based on these traces we can infer the *observable behaviour* [77] of a developer as the sequence of activities performed by the developer on the different software related artifacts. The observable behaviour spans activities across different levels of granularity and characteristics across different dimensions, obtained from different sources.

The next challenge is to *assess the impact of the observed behaviour with respect to an assessment task.* As discussed in Chapter 3, we are often interested in the causes for different kinds of events of interest, such as bug fixes or refactorings. Once we have obtained the different characteristics related to the observable behaviour of a developer, we want to assess how these are related to events of interest, and whether and how they can be best leveraged to gain additional actionable knowledge and insights. The obtained knowledge and insights can be used to improve the outcome of activities performed by a developer, e.g. to lower the chances of causing an event of interest, such as a bug fix.

As software artifacts and their characteristics change over time, developers also gain more experience, become more (or less) involved in a project (or different parts of a project). Consequently, the behaviour of the developers is expected to change as well. Thus, the third challenge is to *determine whether changes in the behaviour of developers can be observed and identified, and whether they have an impact on assessment-related outcomes*. Furthermore, the role of the temporal circumstances of an activity, and not just the activity in isolation, need to be investigated as well.

The fourth challenge is to *investigate transfer opportunities — between different developers within the same project, between different time periods for the same developer* *within the same project, as well as across projects — between different developers involved in different projects and for the same developer involved in different projects.* Transferring behaviour models to different contexts can provide early insights in new projects for which there is no sufficient data available for assessment, or in existing projects when new developers join the project.

The results from developer-centric assessment can be incorporated into automated personalised recommendation systems. Such systems can provide suggestions applicable to the individual developers that target their strengths and weaknesses, rather than generic suggestions that many developers may feel are not applicable to their own way of working. Alternatively, generic suggestions may be prioritised depending on their applicability to a particular developer. The results may also be used to trigger alerts and send personalised messages to developers or managers in case of anomalies deviating from typically observed developer behaviour. The ultimate goal is to better understand how each developer works and produce personalised knowledge that can be directed to those who need it the most in order to make the development process more efficient, rather than flood everyone with generic information that often may not be relevant or useful to them in a given situation. The better understanding of the behaviour of each developer can indicate which kinds of activities can be considered risky in a given context, and incentivise activities that can compensate for potential risks. Such understanding can also be used for taking targeted organisational quality assurance measures, such as specific training sessions, or team reorganisation, where applicable. This kind of understanding lays down the foundation for a change in perspective from broad organisation-wide measures designed for normalised and conforming behaviour to developer- and team-specific measures embracing the strengths and weaknesses arising from different ways of working of each individual developer. This change in perspective and better understanding of the behaviour of individual developers is long overdue given the shift to globally distributed software development both in industrial and open source contexts over the past decades.

4.1.1. Situational and Dispositional Factors

The characterisation of developer behaviour is inspired in part by the notions of dispositional and situational factors determining the behaviour of an individual in a given context, discussed in the human social psychology literature [71]. With a comprehensive characterisation of the circumstances in which a development activity is taking place, as well as potential consequences resulting from the activity, we are concerned with identifying the factors that contribute to the causes for such consequences in a given context. Thus, conceptually we are contemplating the circumstances of a development activity in relation to these notions as defined by:

- the situation in which the activity takes place, described by the artifacts on which the activity was performed and their characteristics at the point in time in which the activity was performed (*situational factors*),

- the developer that performed the activity, described by characteristics pertaining to their experience up to the point in time in which the activity was performed (*dispositional factors*),
- the activity that was performed, described by changes in the characteristics of the artifacts, as well as relations between the characteristics of the artifacts and the characteristics of the developers (*activity factors*),
- the outcome of the activity described by the characteristics of the artifacts after the activity was performed, and in particular characteristics related to an assessment task, such as the presence of a defect or the increase in technical debt as the result of an activity (*consequences*).

The artifacts considered in the characterisation of developer behaviour include primarily the building blocks comprising the software, at different levels of granularity, such as projects, components, packages, files, classes, methods. However, other software related artifacts can be contemplated as well, including issue reports, requirements specifications, test cases and test specifications, mailing list and forum discussions. These may be considered in relation to the behaviour of developers on the building blocks of the software — e.g. activities performed on a file in relation to an issue report, or changes in the test coverage of a method as a result of an activity performed by a developer. Such artifacts may also be considered in relation to activities performed on them directly, where they comprise the primary situational factors. For example, contribution behaviour of developers in an online forum can be characterised with respect to certain topics of interest or also with respect to the roles of individual developers in the forum (in a similar manner as in Sudau et al. [171]). In this case, the behaviour of the developers is considered with respect to the circumstances related to their activities on such artifacts. This can provide us with a broader perspective on the characterisation of developer behaviour.

Consider the conceptual overview for an example scenario illustrating these notions shown in Figure 4.1. Contemplating the behaviour of a developer *joe* that works (or performs development activities) on artifacts a and b at two different points in time t and $t+2$. At time point t, developer *joe* is in state s_t^{joe}, reflecting *joe*'s experience and knowledge at that time. It is indicative of the dispositional factors in the given circumstances of the activities in question. The artifacts a and b are also in states s_t^a and s_t^b, respectively, reflecting their characteristics at that at the time of the activities. They are indicative of the situational factors in the given circumstances of the activities in question. As a result of the activities, the developer gains new experience and knowledge. The next time *joe* performs some development activities, this new experience and knowledge indicative of changed dispositional factors is reflected in s_{t+2}^{joe}. As an outcome of the activities performed by *joe* at time t, the characteristics of the

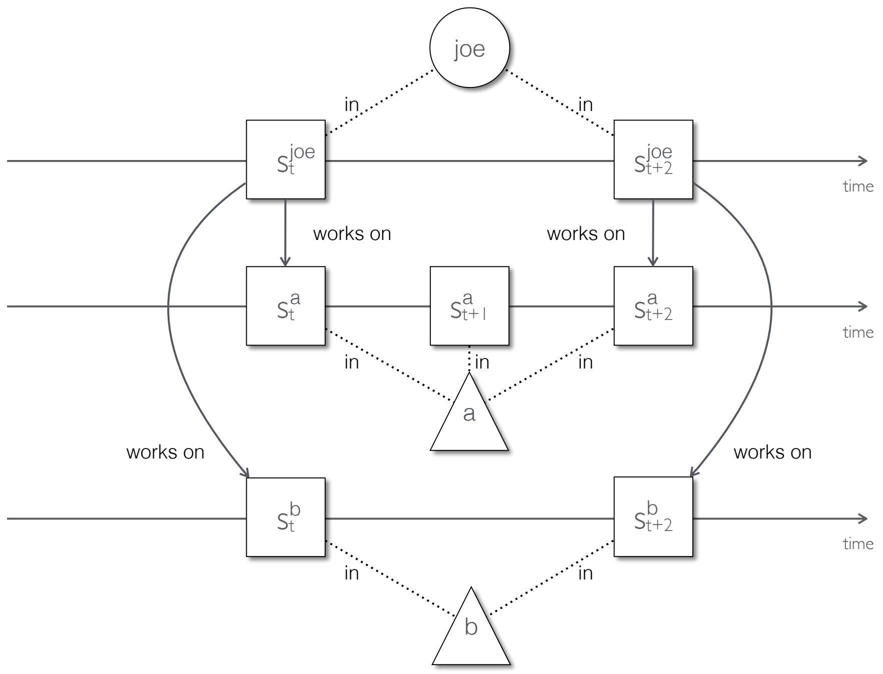


Figure 4.1.: Behavior characterisation conceptual overview

artifacts on which they were performed have also changed. These are reflected in states s^a_{t+1} and s^b_{t+2} resulting from these activities, which are indicative of the changed situational factors that developers performing subsequent activities on these artifacts will be confronted with. Changes to certain characteristics, such as the presence of defects or increase in technical debt are of particular interest for assessment tasks. When at time point $t+2$, developer *joe* performs new activities on these artifacts, *joe* has new experience and is confronted with a new context. In these new circumstances, the activities performed by *joe* at time point $t+2$ may have different consequences with regard to characteristics of interest, even if the actual activities are identical in scope and content.

While the approach discussed in this chapter is conceptually inspired from human social psychology, the current scope of the approach is restricted to characteristics that can be automatically measured from software-related assets, such as VCSs, ITSs, mailing lists, etc. There is already large body of work focusing on personality types and related characteristics of software developers [25, 26, 33, 87, 179] which are relying on data gathered through interviews and questionnaires. However, these require manual intervention, possibly also at different points in time. Such characteristics may be integrated with the characteristics discussed in this chapter as part of future work.

4.1.2. Collaboration Factors

In the example illustrated in Figure 4.1, at time point $t+1$ there is different developer that performed an activity on artifact a in the meantime thus changing its characteristics again between time points $t+1$ and $t+2$. This is where collaboration comes into play. Collaboration characteristics describe interactions between developers. Collaboration characteristics are reflected both in the situational factors — e.g. how many developers have worked on a given artifact up to a given point in time, and in the dispositional factors — e.g. how many developers has a given developer collaborated with up to a given point in time. The characterisation of collaboration behaviour is based on the premise illustrated in Figure 4.2. Given:

- a developer *joe* in a state s_t^{joe} at a time t that performed an activity on an artifact a at a time t resulting in a state s_t^a of a,
- a developer *tom* in a state s_{t+1}^{tom} at a time $t+1$ that performed an activity on the artifact a at a time $t+1$ resulting in a state s_{t+1}^a of a, and
- a developer *pat* in a state s_{t+2}^{pat} at a time $t+2$ that performed an activity on the artifact a at a time $t+2$ resulting in a state s_{t+2}^a of a,

we state that:

- developer *tom* in state s_{t+1}^{tom} directly collaborated with developer *joe*, since *tom* worked on a state s_t^a of artifact a as left by *joe*, thus producing state s_{t+1}^a, and
- developer *pat* in state s_{t+2}^{pat} directly collaborated with developer *tom*, since *pat* worked on a state s_{t+1}^a of artifact a as left by *tom*.

We can then define the *direct collaboration* (*dc*) relationship between two developers d and c, where d is said to have directly collaborated with c, iff for any given artifact a up to a given time point t, there exists a state s_i^a authored[15] by developer d, such that the previous state s_{i-1}^a of that artifact was authored by developer c. That is:

$$d \xrightarrow{dc} c = \{s_i^a \in S_t^a : author(s_i^a) = d \wedge author(s_{i-1}^a) = c\} \tag{4.1.1}$$

Since artifact states build on each other, that is each state of the artifact is typically not completely overwritten by the corresponding author, but rather only partially modified, we also define the *indirect collaboration* (*ic*) relationship between developers d and c. It complements the *direct collaboration* relationship and accounts for the fact that developers are also at least partially exposed to the contributions of all other developers that have performed activities on a given artifact a up to a time point t. Given two

[15]A state of an artifact is *authored* by a developer if it is the result of an activity performed on the artifact by that developer.

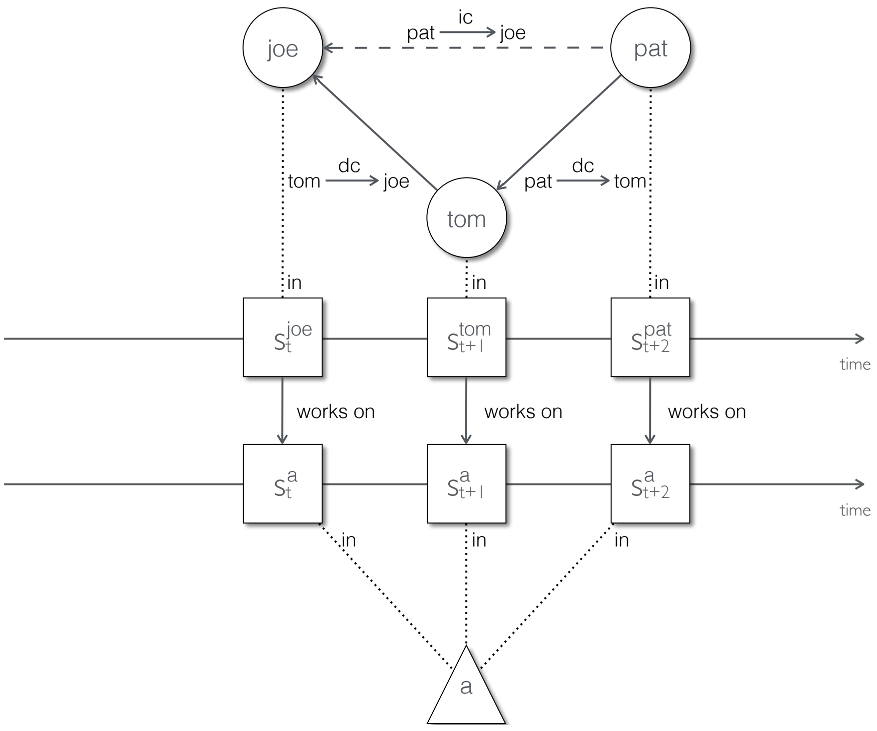


Figure 4.2.: Collaboration characteristics conceptual overview

developers d and c, d is said to have indirectly collaborated with c, iff for any given artifact a up to a given time point t, there exists a state s_i^a authored by developer d, such that an earlier state s_j^a of that artifact was authored by developer c, that is:

$$d \xrightarrow{ic} c = \{s_i^a, s_j^a \in S_t^a : author(s_i^a) = d \wedge author(s_j^a) = c \wedge i > j\} \tag{4.1.2}$$

Indirect collaborations include direct collaborations by definition. In Figure 4.2, the *indirect collaboration* relationship is illustrated by means of a directed dashed line with an arrowhead between *pat* and *joe*, whereas the *direct collaboration* is illustrated by means of directed solid lines with arrowheads between *pat* and *tom*, and between *tom* and *joe*. The indirect collaborations between *pat* and *tom*, and between *tom* and *joe* are not shown explicitly as indirect collaborations also include direct collaborations.

4.2. Dimensions and Characteristics

The conceptual framework for the characterisation of developer behaviour described in Section 4.1 lays down the foundations of our approach towards reasoning about development activities and developer behaviour. In the following sections we contemplate a non-exhaustive list of characteristics describing both the situational and dispositional factors related to the behaviour of developers. The characteristics are tentatively categorised across different dimensions. However, a definitive categorisation is hard to achieve as some characteristics fit into multiple dimensions.

Most of the characteristics are language- and technology-agnostic. They depend to an extent on the granularity and scope of the information provided by the corresponding assets the characteristics are derived from, such as the peculiarities of the VCS, ITS, mailing list or online forum platform. Still, they are applicable to any project making use of a particular VCS, ITS, mailing list or online forum platform.

Some of the characteristics, such as ones based on static analysis results, are specific to a particular language and potentially also a particular tool. While there are a number of measurable characteristics described in the literature [30, 73], different tools may implement only a subset of these. We exemplify several characteristics based on static analysis results provided by a specific tool (*InFamix*[16]) which cover two popular languages (*Java* and *C++*). Similar characteristics can be derived for other languages based on corresponding static analysis results, which can then be integrated into the overall framework for characterising developer behaviour.

Another multi-faceted aspect of characteristics is granularity. From a temporal perspective, granularity has to do with the level of detail in terms of how frequently measurements are (or can be) obtained — hourly, daily, weekly, monthly, yearly, per release, per revision, per key press. Depending on the level of granularity of the intended assessment, measurements may need to be aggregated or distributed. This is of particular interest when it comes to the granularity of developer states, where we can contemplate the state of a developer after every single activity (micro-granularity) or the state of the developer between major milestones in the developer's experience (macro-granularity). The latter may be based on a linear or non-linear scale. It may also be based on clustering or regression models identifying different modes of operation of a developer or different stages based on experience and contribution behaviour.

Similarly, from a spatial perspective granularity has to do with the level of detail of artifacts — methods, classes, packages, files, components, projects. Measurements at different levels may again need to be aggregated into coarser levels of granularity or distributed into finer levels of granularity (see also Section 3.5) depending on the level at which the measurements are available. Adequate aggregation and distribution strategies need to be put in place where applicable.

[16]The tool was discontinued in 2016.

4.2.1. Static Analysis

As noted in Section 4.1, characteristics of the artifacts themselves are usually the first to be utilised for software assessment (artifact-centric). With static analysis tools readily available for most popular languages and technologies, such characteristics are often systematically collected within organisations and within projects, although the temporal granularity may vary, e.g. daily, weekly, monthly, yearly, per release. For a detailed characterisation of developer behaviour, such characteristics need to be obtained with every artifact state, e.g. for every revision. This can be achieved also retrospectively, provided adequate tools for retrospective analysis are available. Static analysis usually also provides structural information at the logical levels of granularity, including the location of classes and methods within physical artifacts such as files, as well as relationships among them.

We make use of static analysis results as both situational and dispositional characteristics. On the one hand, the static analysis results characterise the states of artifacts directly. On the other hand, developers are indirectly characterised by the states of artifacts they have performed activities on. For example, a developer that typically works with small artifacts may be more likely to increase technical risks when working on a large artifact, as that can be considered an anomaly with respect to typical behaviour. The behaviour model can then be calibrated based on observed behaviour for the developer in order to identify which anomalies lead to technical risks.

4.2.1.1. Situational Factors

A number of measurable characteristics have been described in the literature under the umbrella term “software metrics" [30, 73, 103, 119], where object-oriented metrics in particular have been widely accepted as means for quantitative characterisation of artifacts. We contemplate a subset of these that were available as the result of static analysis provided by the tooling we relied on. The metrics-based characteristics are summarised in Table 4.1.

The metrics listed in Table 4.1 serve as an example, based on a particular language and tooling of interest. As noted above, other tools and languages may provide different sets of metrics. It is not the main focus of this chapter to argue which sets of metrics are most appropriate. The literature has come up with various sets of metrics for different purposes and we also argue that the exact metric selection may vary based on the assessment task of choice. In order to obtain sufficiently detailed data for the characterisation of each activity performed by a developer, we extract metrics for each known state of each artifact.

Another kind of static analysis that is frequently applied in practice and has been extensively discussed in the literature is clone (or duplicate) detection [6, 9]. Similar to software metrics, there are various notions on what constitutes duplicated code and dif-

Short Name	**Granularity**	**Full Name**
MX.AMW	Class	Average Method Weight
MX.BOvR	Class	Base Class Overriding Ratio
MX.BUR	Class	Base Class Usage Ratio
MX.CBO	Class	Coupling Between Objects
MX.CPFD	Class	Capsules Providing Foreign Data
MX.CW	Class	Class Weight
MX.DIT	Class	Depth of Inheritance Tree
MX.LCC	Class	Loose Class Cohesion
MX.LCOM	Class	Lack of Cohesion in Methods
MX.NAS	Class	Number of Added Services
MX.NOA	Class	Number of Attributes
MX.NOACCM	Class	Number of Accessor Methods
MX.NOAM	Class	Number of Abstract Methods
MX.NOCHLD	Class	Number of Children
MX.NOM	Class	Number of Methods
MX.NOPRTA	Class	Number of Protected Attributes
MX.NOPRTM	Class	Number of Protected Methods
MX.NOPUBA	Class	Number of Public Attributes
MX.NOPUBM	Class	Number of Public Methods
MX.NOVRM	Class	Number of Overriding Methods
MX.PNAS	Class	Percentage of Newly Added Services
MX.RFC	Class	Response for Class
MX.SPIDX	Class	Specialisation Index
MX.TCC	Class	Tight Class Cohesion
MX.WOC	Class	Weighted Operation Count
MX.StartLine	Method	Starting Line
MX.EndLine	Method	Ending Line
MX.ALD	Method	Access to Local Data
MX.ATFD	Method	Access to Foreign Data
MX.CYCLO	Method	McCabe's Cyclomatic Number
MX.DR	Method	Dispersion Ratio
MX.ICDO	Method	Incoming Coupling Dispersion for an Operation
MX.ICIO	Method	Incoming Coupling Intensity for an Operation
MX.LDA	Method	Locality of Data Accesses
MX.LOC	Method	Lines of Code
MX.LOCOMM	Method	Lines of Comments
MX.MAXNESTING	Method	Maximum Nesting Level of Instructions
MX.NOAV	Method	Number of Accessed Variables
MX.NOOC	Method	Number of Outgoing Calls
MX.NOPAR	Method	Number of Parameters
MX.OCDO	Method	Outgoing Coupling Dispersion for an Operation
MX.OCIO	Method	Incoming Coupling Dispersion for an Operation

Table 4.1.: Object oriented metrics used as situational factors.

Short Name	Granularity	Full Name
DD.NCL	File, Class, Method	Number of Cloned Lines
DD.NCF	File, Class, Method	Number of Cloned Fragments
DD.PCL	File, Class, Method	Percentage of Cloned Lines
DD.CFR	File, Class, Method	Clone Fragment Ratio
DD.VCL	File, Class, Method	Variance of Cloned Lines
DD.ALPF	File, Class, Method	Average Cloned Lines per Cloned Fragment
DD.MLPF	File, Class, Method	Mean Cloned Lines per Cloned Fragment
DD.VLPF	File, Class, Method	Variance of Cloned Lines per Cloned Fragment
DD.CDR	File, Class, Method	Clone Dispersion Ratio

Table 4.2.: Duplication information as situational factors.

ferent approaches to detecting duplicated code discussed in the literature [44, 99, 155]. The available information resulting from the application of duplicate detection may vary depending on the language and tooling. It is not the main focus of this chapter to argue which approach to duplicate detection is most appropriate. Rather, we resort to relying on basic indicators of the amount and location of duplicated code as characteristics for situational factors. For example, we contemplate whether an activity involved creating, modifying, or removing duplicated code, as well as the size and proportion of the duplicated code with respect to the size of overall changes and the corresponding artifacts. The duplication-related characteristics are summarised in Table 4.2. All duplication-related characteristics are applicable on all granularity levels. At the logical level of granularity, duplicated code may span beyond the boundaries of the logical artifact defined by its starting and ending line. In such cases, only the duplicated code within the boundaries of the logical artifact is considered.

Code duplication is also referred to as a “code smell” [50]. Detection of code smells can be used to enrich the description of the context in which an activity was performed and the activity itself even further. In this chapter we contemplate code duplication as an example. Further code smells can be added as needed, depending on the assessment task, and also depending on the availability of tool support for a given language.

While code smells are considered “anti-patterns” in software development describing poor development practices that often lead to increased technical debt, design patterns [49] describe best practices in software development that outline design solutions to common problems. The presence or absence of such design patterns can be used to provide additional description of the context in which an activity was performed and of the activity itself.

The application of bag-of-words [145] and characteristic vectors [83] in software assessment represents light-weight approaches to static analysis. Bag-of-words relies on frequency distributions of individual words in the content of an artifact, thus highlighting the prevalence of referencing particular entities within that artifact. It can be applied on any artifact that has textual content, including commit messages in a VCS, issue descriptions and comments in an ITS, postings in user forums, and messages in mailing lists. An example for the application of bag-of-words is shown in Table 4.3. The text from the previous two paragraphs has been processed for illustrative purposes. *Paragraph 1* refers to the paragraph starting with "Code duplication is also referred to as ..." and *Paragraph 2* refers to the paragraph starting with "While code smells are considered...". During the processing, common terms, such as articles and prepositions, referred to as "stop words" have been excluded. Both paragraphs are combined to create a common list of words. The table includes all the words occurring more than once in both paragraphs combined, with a breakdown for each paragraph. If we contemplate each paragraph individually, the list of the most frequent words will be different, thus it will be difficult to compare them. Instead, relying on the occurrences of all words across the individual paragraphs provides a common ground for comparison against a shared vocabulary. The bag-of-words approach is also the foundation for concept-space analysis [29], term-document matrices [101], and topic models [16], which are used for identifying the prevalence of concepts and the main topics of natural-language documents. Based on the occurrence counts in the example, we can infer that *Paragraph 1* has more to do with the concepts *code*, *duplication*, and *smells*, whereas *Paragraph 2* is concerned more with the concepts *patterns*, *design*, *development*, and *practices*. In addition, stemming is frequently used to remove morphological variations. More sophisticated approaches consider also word co-occurrences, such as *code smells* and *design patterns*, which provide more detailed insights about the concepts used. The bag-of-words approach is used in a similar manner for identifying the concept-space of source code [137], so that frequent mentions of concepts from the GUI domain (such as references to GUI libraries) can be used to infer that a class or a method deals with GUI-related functionality, for example. Apart from absolute occurrence counts, bag-of-words may also rely on density measurements which relate the number of occurrences to the size of the document or the size of all the documents.

A characteristic vector is similar to a bag of words, but instead of using the content directly, it relies on a higher level syntactical representation of the content. Thus, instead of the frequencies of the textual tokens, it considers the frequencies of the corresponding *Abstract Syntax Tree* (AST) nodes, where the level of the AST nodes may also vary. As such, characteristic vectors are better suited for source code artifacts rather than natural language. However, while they provide more structural information, due to the abstraction they also lose domain information, since all identifiers are represented by the same type of AST nodes.

Characteristic vectors were first used in software assessment for the application of defect prediction in [84]. For the evaluation, the authors considered only the characteristic vector after a change and the difference to the characteristic vector before the

Word	Both Paragraphs	Paragraph 1	Paragraph 2
code	5	4	1
activity	4	2	2
design	3	0	3
development	3	0	3
patterns	3	0	3
smells	3	2	1
duplication	2	2	0
practices	2	0	2
software	2	0	2
depending	2	2	0
context	2	1	1
description	2	1	1
perform	2	1	1
used	2	1	1

Table 4.3.: Bag-of-words example.

change. As we are contemplating the situational factors describing the context where an activity takes place, it makes sense to also consider the characteristic vector before the change in our context. The same applies to bag-of-words. While the number of AST nodes is naturally limited by the grammar of the language, the number of words can be arbitrarily long. For characteristic vectors, it often makes sense to focus on AST nodes representing constructs of particular interest, such as loops and conditional statements, which can be used to reduce the number of characteristics being considered. In the bag-of-words approach, it is not as straightforward to determine adequate threshold for the number of words to be considered. It can be based on minimum occurrence count (e.g. words occurring more than once), ranking (e.g. top ten most frequently occurring words), or other constraints.

For all characteristics discussed above, we calculate the differences between the values in the state on which an activity was performed and the state resulting from the activity. These differences characterise the activity itself (activity factors).

4.2.1.2. Dispositional Factors

We extrapolate a set of characteristics describing the state of a developer at a given point in time based on the available artifact and activity characteristics. The measurements obtained from the static analysis of the individual states of the artifacts a developer has worked on, as well as the corresponding deltas are considered indicative of the experience of the developer. In order to capture this aspect of the behaviour, we include characteristics for the distribution (mean, standard deviation) of the static analysis measurements and deltas in each developer state. These characteristics can be further refined into:

- *lifetime distribution* encompassing the distribution of the characteristics obtained from measurements collected throughout the complete sequence of activities performed by a developer, and
- *sliding window distribution* encompassing the distribution of the characteristics obtained from measurements collected from a subset of the activities performed by a developer, limited to the most-recent activities determined either by the last n activities, by the activities within the time period t, or by activities within the last n states of the developer.

In order to capture the relation between the situational and dispositional situational factors, we can also keep track of the distances between the situational and corresponding dispositional factors (mean in this case), or the distances between the activity factors and the corresponding dispositional factors (standard deviation in this case). These can serve as an indication of the typical operational ranges for a developer, where larger deviations are indicative of unusual behaviour. Consider the example shown in Figure 4.3 which illustrates some of the static analysis characteristics applied to the conceptual example from Figure 4.1. For illustrative purposes, we assume that the dispositional characteristics for *joe* in s_t^{joe} reflect the experience after five activities with $m.MX.LOC = 32$ (mean LOC) and $sd.MX.LOC = 6$ (standard deviation for LOC). The characteristics of the activities describing *joe*'s work on a (shown in gray) include:

- the difference between the measurements for the characteristic LOC in the state of a on which *joe* was working and the state of a resulting from *joe*'s work ($d.MX.LOC = 5$),
- the difference between the measurements for the characteristic LOC in the state of a on which *joe* was working and *joe*'s mean for this characteristic so far ($d.m.MX.LOC = 7$), which in this case indicates that the artifact is smaller than the average size of artifacts that *joe* is working on,
- the absolute difference between $d.MX.LOC$ and *joe*'s standard deviation for the LOC characteristic so far ($d.sd.MX.LOC = 1$), which in this case suggests that the difference from the average size of artifacts is close to the range of variation with respect to the size of artifacts within which *joe* is typically working.

In comparison, the corresponding characteristics of the activities describing *joe*'s work on b are $d.MX.LOC = 8$, $d.m.MX.LOC = 18$, $d.sd.MX.LOC = 2$. After these activities, the corresponding dispositional characteristics for *joe* in s_{t+2}^{joe} have changed correspondingly to reflect the new experience collected by *joe*. Similar characteristics are calculated for the remaining static analysis measurements. While we only illustrated the use of the mean and standard deviation here, other distributional characteristics,

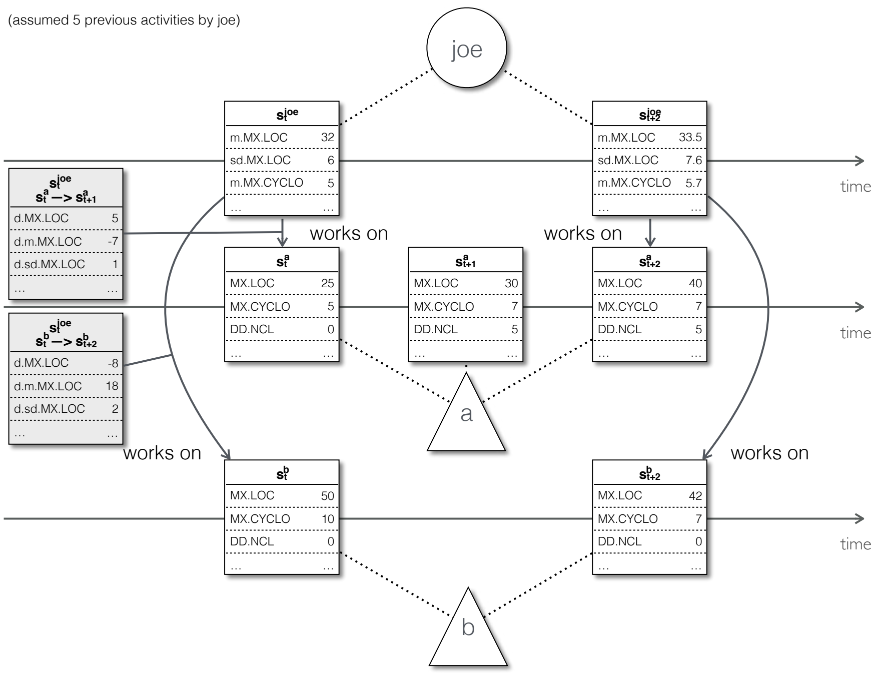


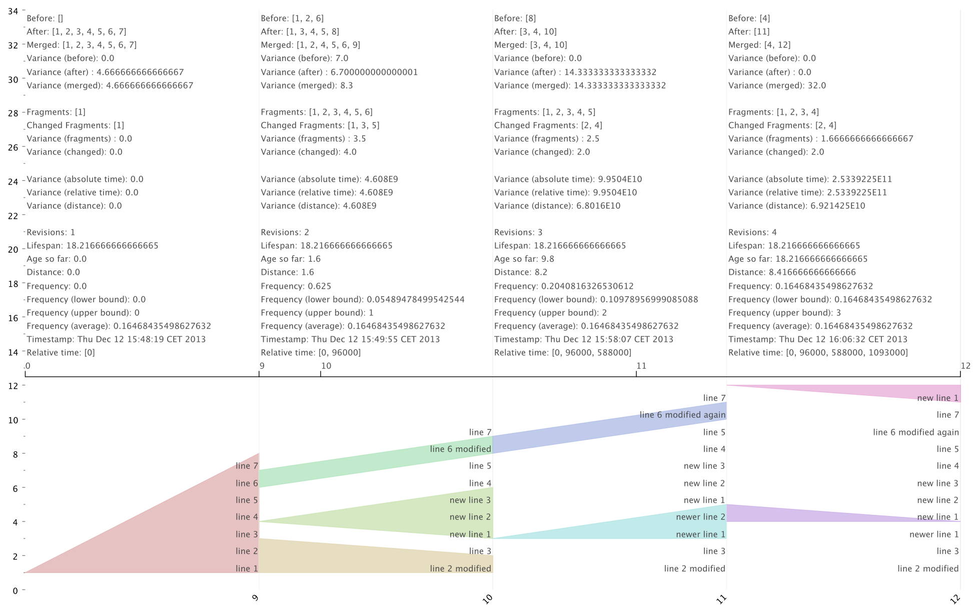
Figure 4.3.: Example characterisation based on static analysis

such as minimum, maximum, etc., can also be considered for the characterisation of dispositional factors.

4.2.2. Spatial

The spatial characteristics are concerned with the location of changes within an artifact. They are based on the changed lines in a given state of an artifact. The spatial characteristics are indicative of the dispersion of changes. Both the lines before the change and the lines after the change are considered in separate sets. The *lines before* (L_{before}) denote the set of lines of the previously recorded revision of an artifact that were modified or deleted in a given revision. The *lines after* (L_{after}) denote the set of lines in the revision that were modified or added in that revision. Thus, the lines before lack any information regarding added lines that were not part of a modification, while the lines after lack any information regarding removed lines that were not part of a modification.

For illustrative purposes, we will rely on an annotated representation of a line change map, similar to the ones discussed in Chapter 3. An example of such an annotated line change map is shown in Figure 4.4. As with previous line change maps, revisions are represented as segments on the x-axis and lines are represented on the y-axis. Coloured



```
Before: []
After: [1, 2, 3, 4, 5, 6, 7]
Merged: [1, 2, 3, 4, 5, 6, 7]
Variance (before): 0.0
Variance (after) : 4.666666666666667
Variance (merged): 4.666666666666667

Fragments: [1]
Changed Fragments: [1]
Variance (fragments) : 0.0
Variance (changed): 0.0

Variance (absolute time): 0.0
Variance (relative time): 0.0
Variance (distance): 0.0

Revisions: 1
Lifespan: 18.216666666666665
Age so far: 0.0
Distance: 0.0
Frequency: 0.0
Frequency (lower bound): 0.0
Frequency (upper bound): 0
Frequency (average): 0.16468435498627632
Timestamp: Thu Dec 12 15:48:19 CET 2013
Relative time: [0]
```

```
Before: [1, 2, 6]
After: [1, 3, 4, 5, 8]
Merged: [1, 2, 4, 5, 6, 9]
Variance (before): 7.0
Variance (after) : 6.700000000000001
Variance (merged): 8.3

Fragments: [1, 2, 3, 4, 5, 6]
Changed Fragments: [1, 3, 5]
Variance (fragments) : 3.5
Variance (changed): 4.0

Variance (absolute time): 4.608E9
Variance (relative time): 4.608E9
Variance (distance): 4.608E9

Revisions: 2
Lifespan: 18.216666666666665
Age so far: 1.6
Distance: 1.6
Frequency: 0.625
Frequency (lower bound): 0.05489478499542544
Frequency (upper bound): 1
Frequency (average): 0.16468435498627632
Timestamp: Thu Dec 12 15:49:55 CET 2013
Relative time: [0, 96000]
```

```
Before: [8]
After: [3, 4, 10]
Merged: [3, 4, 10]
Variance (before): 0.0
Variance (after) : 14.333333333333332
Variance (merged): 14.333333333333332

Fragments: [1, 2, 3, 4, 5]
Changed Fragments: [2, 4]
Variance (fragments) : 2.5
Variance (changed): 2.0

Variance (absolute time): 9.9504E10
Variance (relative time): 9.9504E10
Variance (distance): 6.8016E10

Revisions: 3
Lifespan: 18.216666666666665
Age so far: 9.8
Distance: 8.2
Frequency: 0.2040816326530612
Frequency (lower bound): 0.10978956999085088
Frequency (upper bound): 2
Frequency (average): 0.16468435498627632
Timestamp: Thu Dec 12 15:58:07 CET 2013
Relative time: [0, 96000, 588000]
```

```
Before: [4]
After: [11]
Merged: [4, 12]
Variance (before): 0.0
Variance (after) : 0.0
Variance (merged): 32.0

Fragments: [1, 2, 3, 4]
Changed Fragments: [2, 4]
Variance (fragments) : 1.6666666666666667
Variance (changed): 2.0

Variance (absolute time): 2.5339225E11
Variance (relative time): 2.5339225E11
Variance (distance): 6.921425E10

Revisions: 4
Lifespan: 18.216666666666665
Age so far: 18.216666666666665
Distance: 8.416666666666666
Frequency: 0.16468435498627632
Frequency (lower bound): 0.16468435498627632
Frequency (upper bound): 3
Frequency (average): 0.16468435498627632
Timestamp: Thu Dec 12 16:06:32 CET 2013
Relative time: [0, 96000, 588000, 1093000]
```

Figure 4.4.: Spatial characteristics example (see Figure A.4 for a larger version)

blocks highlight changed fragments between revisions (within a revision's segment). In addition, an overlay shows the actual content of the lines of a file in each revision (near the right hand side of a segment on the x-axis). This is mostly for illustrative purposes. For artificially constructed examples the content may still be readable, but in a file from an actual software system with a large number of revisions, this overlay will quickly become illegible. Above the changed lines, an overlay shows additional data points related to the spatial and temporal characteristics (Section 4.2.3). Around the vertical middle of the line change map in Figure 4.4, a timeline overlay indicates the relative time point of each revision plotted against the lifetime of the artifact.

Consider the example shown in Figure 4.4. In *revision 10*, $L_{before} = \{1,2,6\}$, thus it lacks information regarding the three added lines in *revision 10*. Note also, that due to additions, deletions, and modifications the location of a given line may shift between revisions, as is the case of *line 6* in *revision 9* which becomes *line 7* in *revision 10*, for example. In order to account for these peculiarities, the *lines merged* (L_{merged}) set is defined. It aims to address some of the concerns above. This set is constructed by taking the lines from both sets (L_{before} and L_{after}) and projecting them on the same domain, while taking into account any offsets where applicable. In particular, for every fragment it takes all lines before the change and to each line it adds an offset (O_{after}) resulting from an increase in the number of lines after modifications or additions in prior fragments of a revision. If the fragment only adds lines, then the set takes all the lines after the change and to each line it adds an offset (O_{before}) resulting from a decrease to the number of lines after modifications or deletions in prior fragments of a revision. Consider, for example, *revision 10*, given ($L_{before} = \{1,2,6\}$ and $L_{after} = \{1,3,4,5,8\}$, we contemplate the following fragments:

- *lines 1* and *2* are changed in that *line 1* is removed and *line 2* is modified;
- three new lines are added after *line 3*, which effectively becomes *line 2* after *line 1* is removed in the preceding fragment, where the new lines become *lines 3*,*4*, and *5* after the revision;
- in a third fragment, *line 6* is modified, and because of the offset resulting from the removal of *line 1* and the addition of the three new lines, it effectively becomes *line 8* after the revision.

The result is a sequence of operations on lines, which transform an artifact from one state into another within a revision. Table 4.4 summarises the application of this approach on *revision 10*. The *Merged* column plots all line on the same domain following the approach described above. The *Before* column maps the lines before the change to the merged lines domain, whereas the *After* column maps the lines after the change to the merged lines domain and to the corresponding lines before the change. Finally, the *Change* column indicates the whether a line was *added (+)*, *modified (*)*, *removed (-)*, or *preserved ( )*. Each row corresponds to a line, regardless of whether it is affected by changes in a revision. In terms of change operations, the *Merged* column indicates the sequential number of each line operation (add, remove, modify, preserve) and the change column indicates the type of operation. In this case, L_{merged} includes the lines from the *Merged* column for which there is an operation other than preserve, resulting in $L_{merged} = \{1,2,4,5,6,9\}$. Based on the three sets, we calculate the variance of each set as a spatial characteristic.

The spatial characteristics can also be used to approximate recurring changes in an artifact, affecting the same lines or lines in close proximity. Consider the example shown in Figure 4.5. In *revision 2*, *lines 3*, *5*, and *7* are modified. The subsequent *revision 3* adds three new lines between *line 1* and *line 2*. Then, in *revision 4*, the same lines from *revision 3* ($\{3,5,7\}$) are modified again, but because of the three new lines introduced in *revision 3*, their location has now shifted to $\{6,8,10\}$. The variance in this case remains the same, thus it can serve as an indication that there was a recurring change in same location. If in this case *revision 2* is considered as the cause for an event of interest, the fact that *revision 4* is identical with respect to the spatial characteristics can be used to guide further inspection of the changes in *revision 4*. Similar to the line-based spatial characteristics, we also obtain fragment-based spatial characteristics, which reflect the dispersion of changes at a coarser level of granularity.

Merged	Before	Change	After
1	1	-	
2	2	*	1
3	3		2
4		+	3
5		+	4
6		+	5
7	4		6
8	5		7
9	6	*	8
10	7		9

Table 4.4.: Line merging example

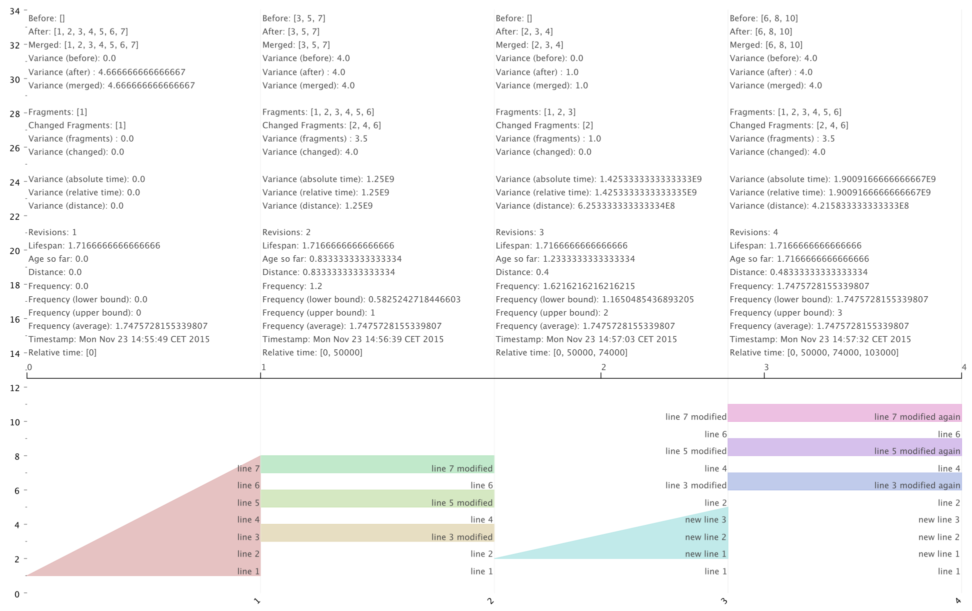
Short Name	Granularity	Full Name
SP.NLB	File, Class, Method	Number of Changed Lines (before)
SP.NLA	File, Class, Method	Number of Changed Lines (after)
SP.NLM	File, Class, Method	Number of Changed Lines (merged)
SP.VLB	File, Class, Method	Variance of Changed Lines (before)
SP.VLA	File, Class, Method	Variance of Changed Lines (after)
SP.VLM	File, Class, Method	Variance of Changed Lines (merged)
SP.NCF	File, Class, Method	Number of Changed Fragments
SP.VCF	File, Class, Method	Variance of Changed Fragments
SP.CM1	File, Class, Method	M1 from [127]: Churned LOC / Total LOC
SP.CM2	File, Class, Method	M2 from [127]: Deleted LOC / Total LOC

Table 4.5.: Spatial information as situational factors.

4.2.2.1. Situational Factors

The spatial characteristics are used primarily for the characterisation of the situational factors. An overview of the situational factors based on the spatial characteristics is shown in Table 4.5. The characteristics at the logical levels of granularity (Class, Method) need to be interpolated within the scope of the corresponding artifact. The number of changed lines before (*SP.NLB*) and after (*SP.NLB*) are frequently used in the literature as the basis for the so-called *churn* metrics [91, 127]. We consider churn metrics as part of the spatial characteristics. We adopt two of them ($SP.CM1$,$SP.CM2$) for the characterisation of the situational factors[17].

[17]The original authors used number of non-commented executable lines, we use the total number of lines, including comments and blank lines.



```
Before: []
After: [1, 2, 3, 4, 5, 6, 7]
Merged: [1, 2, 3, 4, 5, 6, 7]
Variance (before): 0.0
Variance (after) : 4.666666666666667
Variance (merged): 4.666666666666667

Fragments: [1]
Changed Fragments: [1]
Variance (fragments) : 0.0
Variance (changed): 0.0

Variance (absolute time): 0.0
Variance (relative time): 0.0
Variance (distance): 0.0

Revisions: 1
Lifespan: 1.7166666666666666
Age so far: 0.0
Distance: 0.0
Frequency: 0.0
Frequency (lower bound): 0.0
Frequency (upper bound): 0
Frequency (average): 1.7475728155339807
Timestamp: Mon Nov 23 14:55:49 CET 2015
Relative time: [0]
```

```
Before: [3, 5, 7]
After: [3, 5, 7]
Merged: [3, 5, 7]
Variance (before): 4.0
Variance (after) : 4.0
Variance (merged): 4.0

Fragments: [1, 2, 3, 4, 5, 6]
Changed Fragments: [2, 4, 6]
Variance (fragments) : 3.5
Variance (changed): 4.0

Variance (absolute time): 1.25E9
Variance (relative time): 1.25E9
Variance (distance): 1.25E9

Revisions: 2
Lifespan: 1.7166666666666666
Age so far: 0.8333333333333334
Distance: 0.8333333333333334
Frequency: 1.2
Frequency (lower bound): 0.5825242718446603
Frequency (upper bound): 1
Frequency (average): 1.7475728155339807
Timestamp: Mon Nov 23 14:56:39 CET 2015
Relative time: [0, 50000]
```

```
Before: []
After: [2, 3, 4]
Merged: [2, 3, 4]
Variance (before): 0.0
Variance (after) : 1.0
Variance (merged): 1.0

Fragments: [1, 2, 3]
Changed Fragments: [2]
Variance (fragments) : 1.0
Variance (changed): 0.0

Variance (absolute time): 1.4253333333333333E9
Variance (relative time): 1.4253333333333335E9
Variance (distance): 6.253333333333334E8

Revisions: 3
Lifespan: 1.7166666666666666
Age so far: 1.2333333333333334
Distance: 0.4
Frequency: 1.6216216216216215
Frequency (lower bound): 1.1650485436893205
Frequency (upper bound): 2
Frequency (average): 1.7475728155339807
Timestamp: Mon Nov 23 14:57:03 CET 2015
Relative time: [0, 50000, 74000]
```

```
Before: [6, 8, 10]
After: [6, 8, 10]
Merged: [6, 8, 10]
Variance (before): 4.0
Variance (after) : 4.0
Variance (merged): 4.0

Fragments: [1, 2, 3, 4, 5, 6]
Changed Fragments: [2, 4, 6]
Variance (fragments) : 3.5
Variance (changed): 4.0

Variance (absolute time): 1.9009166666666667E9
Variance (relative time): 1.9009166666666667E9
Variance (distance): 4.215833333333333E8

Revisions: 4
Lifespan: 1.7166666666666666
Age so far: 1.7166666666666666
Distance: 0.48333333333333334
Frequency: 1.7475728155339807
Frequency (lower bound): 1.7475728155339807
Frequency (upper bound): 3
Frequency (average): 1.7475728155339807
Timestamp: Mon Nov 23 14:57:32 CET 2015
Relative time: [0, 50000, 74000, 103000]
```

Figure 4.5.: Recurring changes example (see Figure A.5 for a larger version)

4.2.2.2. Dispositional Factors

The dispositional spatial characteristics are derived from the situational spatial characteristics in a way similar to the characteristics based on static analysis. Here, we only contemplate the characteristics based on the number of changed lines and fragments, for which we calculate distributional characteristics indicative of the experience of a developer up to a given point in time.

4.2.3. Temporal

To account for temporal relationships between development activities, we introduce the notion of temporal characteristics. The temporal characteristics are primarily based on distance (time span between subsequent states) and relative time (time span between the first state of an artifact or a developer and a given point in time). In addition to capturing temporal characteristics over all states, we can also project the temporal characteristics over states exhibiting a particular characteristic, such as being identified as causes for events of interest. Since all temporal measurements are originally collected in milliseconds, which can be impractical for feedback and some calculations, we transform them into larger time units such as minutes, hours, and days for a coarser level of temporal granularity. While the temporal granularity of minutes can be useful for the illustrative examples, working with data from real software projects usually requires even coarser level of granularity, such as days. Whichever level of granularity is selected, it should be used throughout all measurements and experiments in order to avoid confusion.

4.2.3.1. Situational Factors

To characterise artifact states temporally, we record the *number of states* (*TM.NOS*) up to a given point in time, the *age* (*TM.AGE*) of the artifact up to that point in time, and the *distance* (*TM.DIST*) to the previous state of the artifact. Based on these direct measurements, we also calculate the *frequency* (*TM.FREQ*) of activities performed on the artifact. In order to establish the temporal dispersion of the activities, we also consider the variance of the age (*TM.VAGE*) and the variance of the distance (*TM.VDIST*) up to the point in time a state was created. Finally, we also contemplate the mean of the distances (*TM.ADIST*) and frequencies (*TM.AFREQ*) up to the point in time a state was created. The example in Figure 4.4 already includes these characteristics, as well as other temporal characteristics used for informative purposes.

The frequency notion is similar, in a sense, to the notion of speed — distance (number of states) over time (age). The intuition is that, if an artifact has high frequency it “moves with a faster speed”, meaning it “travels” a longer “distance” (is subjected to more activities) for a given time period. This potentially entails further consequences associated with “higher speed”, such as higher risk of accidents, while also implying the opposite about lower speeds. The basic concept corresponds to a notion of “average speed”. However, there is also a need for the notion of “current speed”, reflected by the current frequency (*TM.CFREQ*).

Beyond contemplating spatial and temporal characteristics individually, we also identify several characteristics converging on the spatial and temporal domains. These include measures for churn per state of an artifact (*ST.CLS*, *ST.DLS*), but also churn per time period (day, week, month) (*ST.CLP.∗*, *ST.DLP.∗*). An overview of the situational factors based on the temporal characteristics is shown in Table 4.6 and Table 4.7.

4.2.3.2. Dispositional Factors

In addition to the distribution characteristics for the situational temporal and spatio-temporal factors (*m.TM.∗*, *sd.TM.∗*, *m.ST.∗*, *sd.ST.∗*), which are derived in a similar manner to the distribution characteristics for the static analysis situational factors, we also contemplate several temporal characteristics that are based on the states of the developer. We consider the frequency of activities (*TM.DFREQ*), the distance between activities (*TM.DISTA*), the “age” (*TM.DAGE*) of the developer (here we mean the time they have spent on a project rather than their actual human age). As discussed previously, developer states may be determined based on different criteria. Thus, characteristics related to the developer states need to be carefully considered, depending on the criteria and the concrete assessment application. Consequently, we also include the

Short Name	Granularity	Full Name
TM.NOS	File, Class, Method	Number of States
TM.AGE	File, Class, Method	Artifact Age
TM.DIST	File, Class, Method	Distance to Previous State
TM.FREQ	File, Class, Method	Frequency of Activities
TM.VAGE	File, Class, Method	Variance of Artifact Age at States
TM.VDIST	File, Class, Method	Variance of Distances to Previous States
TM.ADIST	File, Class, Method	Mean of Distances to Previous States
TM.AFREQ	File, Class, Method	Mean of Frequency of Activities
TM.CFREQ	File, Class, Method	Current Frequency of Activities

Table 4.6.: Temporal information as situational factors.

Short Name	Granularity	Full Name
ST.CLS	File, Class, Method	Churned Lines per State
ST.DLS	File, Class, Method	Deleted Lines per State
ST.CLP.*	File, Class, Method	Churned Lines per Time Period
ST.DLP.*	File, Class, Method	Deleted Lines per Time Period

Table 4.7.: Spatio-temporal information as situational factors.

characteristics related to the developer's activities which provide more detailed information that is not influenced by the way the developer states are determined. However, activities also have a shortcoming in that multiple activities may occur at the same time, such as multiple artifacts being changed and committed together, especially across multiple levels of granularity. Thus, activity-related characteristics may be further refined across different levels of granularity. An overview of the dispositional factors based on the temporal characteristics is shown in Table 4.8. We do not include the breakdown of activity-related characteristics across multiple levels of granularity here as this depends on the available levels of granularity in a particular context. An application in a particular context may include such a breakdown as necessary.

Short Name	Granularity	Full Name
TM.DNOS	Developer	Number of Developer States
TM.DAGE	Developer	Developer "Age"
TM.DISTS	Developer	Distance to Previous State
TM.DISTA	Developer	Distance to Previous Activity
TM.DFREQ	Developer	Frequency of Activities

Table 4.8.: Temporal information as dispositional factors.

4.2.4. Experience

Experience characteristics are based on the number of contributions and the notion of ownership derived from the size of contributions (based on [57]). Experience characteristics are indicative of the previous knowledge of a developer with respect to a particular artifact or overall.

4.2.4.1. Situational Factors

Experience characteristics are primarily concerned with the contribution experience of developers. As a consequence, experience information that is used as situational factors is projected over the artifact and the developer. Thus, it is attributable to the activity rather than the state of the artifact. The number of contributions of a developer to an artifact (*EXP.CC*) accounts for the domain experience of the developer up to the time of the activity performed on the artifact. A developer with more experience of working on an artifact can be considered more familiar with it. The ratio of the number of contributions by a developer to all contributions to an artifact (*EXP.CCR*) accounts for the relative domain experience of the developer up to the time of the activity performed on the artifact. A developer with more experience of working on an artifact in comparison to other developers can be considered more familiar with it. The average developer contribution count ratio (*EXP.ACCR*) accounts for the diversity of developers contributing to an artifact and serves as a baseline for what can be considered above or below average relative author experience on the artifact. The fractal distribution of the contribution ratios (*EXP.FCCR*) accounts for the diversity of developers contributing to an artifact. It serves as an indication of the distribution of ownership among the developers. A high value would indicate a large number of contributing developers, each contributing to a small proportion of states. A low value would indicate the presence of a major contributor authoring a large proportion of the states. It is based on the notion of fractal value as defined in [35]. The artifact LOC owned by a developer (*EXP.OWN*) is indicative of the amount of code that is known to the developer. While a developer may have made most of the contributions to an artifact, another developer may have modified a large portion of the content of the artifact recently. Thus, the former developer may not be familiar with the current content of an artifact despite the large number of contributions to it. The ratio of artifact LOC owned by a developer (*EXP.OWNR*) indicates the percentage of artifact LOC owned by a developer. The artifact LOC contributed by a developer (*EXP.OWNC*) sums up all the code contributed to an artifact by a given developer up to a given point in time. The contribution retention ratio (*EXP.CRR*) indicates the stability of contributions by a developer. Low retention ratio indicates that contributions by are more likely to be overwritten either by the same developer or by other developers. The developer contribution focus (*EXP.CF*) is based on the number of contributions to an artifact over the total number of contributions by a developer. It

Short Name	Granularity	Full Name
EXP.CC	File, Class, Method	Developer Contribution Count
EXP.CCR	File, Class, Method	Developer Contribution Count Ratio
EXP.ACCR	File, Class, Method	Average Developer Contribution Count Ratio
EXP.FCCR	File, Class, Method	Fractal Distribution of ACCR
EXP.OWN	File, Class, Method	Artifact LOC Owned by Developer
EXP.OWNR	File, Class, Method	Ratio of Artifact LOC Owned by Developer
EXP.OWNC	File, Class, Method	Artifact LOC Contributed by Developer
EXP.CRR	File, Class, Method	Contribution Retention Ratio by Developer
EXP.CF	File, Class, Method	Developer Contribution Focus

Table 4.9.: Experience information as situational factors.

is indicative of the focus of the developer on the artifact. The experience characteristics can also be refined further over the type of activity (e.g. $EXP.CC.*$), such as fix for an issue ($EXP.CC.FIX$), refactoring ($EXP.CC.REF$), etc. An overview of the situational factors based on the experience characteristics is shown in Table 4.9.

4.2.4.2. Dispositional Factors

Dispositional experience characteristics include the distribution characteristics for the situational experience characteristics ($m.EXP.*$, $sd.EXP.*$) and characteristics describing the overall experience of a developer. The distribution characteristics for the situational experience characteristics are derived in a similar manner to the distribution characteristics for the other situational factors. The characteristics describing the overall experience of a developer include the number of activities ($EXP.NOA$), the number of artifacts a developer has worked on ($EXP.NKA$), and the ratio of artifacts a developer has worked on to the total number of artifacts ($EXP.RKA$). The latter characteristics are indicative of the breadth of knowledge and experience of the developer based on working on different artifacts rather than the depth of knowledge while working on the same artifact. The number of activities can also be refined further over the type of activity ($EXP.NOA.*$), such as fix for an issue ($EXP.NOA.FIX$), refactoring ($EXP.NOA.REF$), new feature ($EXP.NOA.NF$), issue report comment ($EXP.NOA.IRC$), etc. We also consider the fractal distribution of $EXP.FCCR$ ($EXP.DCCR$) as an indication of the diversity of the developers that contributed to the state of the artifacts the developer has performed activities on at a certain point in time. Higher diversity is a potential indicator for more heterogeneity in the corresponding artifacts. The overall LOC owned by a developer ($EXP.DOWN$) and the ratio of the owned LOC ($EXP.DOWNR$) are indicative of the overall proportion of code known to a developer. When compared to $EXP.OWN$ and $EXP.OWNR$, these characteristics can be used to infer that a developer is getting in a new territory contributing to an artifact with low-ownership, while having

Short Name	Granularity	Full Name
EXP.NOA	Developer	Number of Developer Activities
EXP.NKA	Developer	Number of Known Artifacts
EXP.RKA	Developer	Ratio of Known Artifacts
EXP.DCCR	Developer	Fractal Distribution of FCCR
EXP.DOWN	Developer	Overall LOC Owned by Developer
EXP.DOWNR	Developer	Ratio of Overall LOC Owned by Developer
EXP.DOWNC	Developer	Overall LOC Contributed by Developer
EXP.DCRR	Developer	CRR for Developer Overall

Table 4.10.: Experience information as dispositional factors.

high overall ownership. Similar to the *EXP.DOWN*, the overall LOC contributed by a developer sums up the amount of code contributed by the developer. The overall contribution retention ratio for a developer (*EXP.DCCR*) is indicative of the overall stability of code contributed by the developer. An overview of the dispositional factors based on the experience characteristics is shown in Table 4.10. We do not include the breakdown of experience-related characteristics across multiple levels of granularity here as this depends on the available levels of granularity in a particular context. An application in a particular context may include such a breakdown as necessary. Here we also do not include dispositional characteristics derived from the distribution characteristics for the situational experience characteristics.

4.2.5. Collaboration

Collaboration characteristics are based on the notions of collaboration discussed in Section 4.1.2. Collaboration characteristics are indicative of the interactions between developers reconstructed based on their work on shared artifacts and overall. While we extrapolate characteristics from the domain of VCSs, the collaboration characteristics discussed below are also applicable to ITSs, mailing lists, and user forums. In this case, the concrete notions of issues, comments, messages, and postings need to be mapped to the more abstract conceptual notions of artifacts and states.

4.2.5.1. Situational Factors

The collaboration characteristics are primarily concerned with relationships between developers. However, since these relationships are derived from shared artifacts on which developers work, some of the collaboration characteristics are also projected on the corresponding artifacts. This provides us with situational information, such as whether the developer working on a given state of an artifact is the same as the developer that worked on the previous state of that artifact (*COL.SAL*), which can indicate a

Short Name	**Granularity**	**Full Name**
COL.SAL	File, Class, Method	Same Author as Last
COL.ACC	File, Class, Method	Artifact Collaborator Count
COL.ACR	File, Class, Method	Artifact Collaborator Ratio
COL.CBLA	File, Class, Method	Collaborations Between Author and Last Author
COL.RBLA	File, Class, Method	Collaboration Ratio Between Author and Last Author

Table 4.11.: Collaboration information as situational factors.

better familiarity of the developer with the content of the artifact since no one else has modified it in the meantime. The number of developers that have collaborated on an artifact up to a given point in time (*COL.ACC*) accounts for the diversity of developers contributing to the artifact. The ratio of developers that have collaborated on an artifact up to a given point in time to the total number of developers up to that point in time (*COL.ACR*) reflects the proportion of all developers that have contributed to an artifact up to a given point in time, i.e., how popular is a given artifact within the population of developers. The number of direct collaborations between the developer working on a given state of an artifact and the developer that worked on the previous state of that artifact (*COL.CBLA*) accounts for the experience of the developer working the target state with working on states of the artifact resulting from activities of the developer working on the previous state. It is an indication of the collaboration between both developers. The developers need not be different, that is if the author of the source state is the same as the author of the target state, it is an indication of how often the developer worked on the artifact as they left it. The collaboration ration between the developer working the target state and the developer working on the previous state (*COL.RBLA*) accounts for the relative collaboration experience of the developer working on the target state with working on states of the artifact resulting from activity of the developer working on the previous state, when contemplating the total collaboration experience of the developer working on the target state. It is an indication of the proportion of activities of the developer on the artifact that followed activities of the developer that worked on the previous state to the total number of activities of the developer on the artifact. The developers need not be different, that is if the author of the source state is the same as the author of the target state, it is an indication of how often the developer worked on the artifact as they left it.

4.2.5.2. Dispositional Factors

Similar to other dispositional characteristics, the dispositional collaboration characteristics include the distribution characteristics for the situational collaboration character-

Short Name	Granularity	Full Name
COL.DCC.*	Developer	Direct Collaborator Count
COL.ICC.*	Developer	Indirect Collaborator Count
COL.DCR.*	Developer	Direct Collaborator Ratio
COL.ICR.*	Developer	Indirect Collaborator Ratio
COL.DIR.*	Developer	Direct to Indirect Collaborator Ratio

Table 4.12.: Collaboration information as dispositional factors.

istics ($m.COL.*$, $sd.COL.*$, with the exception of $COL.SAL$). They also include characteristics describing the overall collaboration of a developer. The number of direct ($COL.DCC.*$) and indirect ($COL.IDC.*$) collaborators account for diversity of developers contributing to artifacts on which a given developer has worked and emphasises the amount of collaborations for a developer. The corresponding ratios ($COL.DCR.*$, $COL.ICR.*$) indicate the proportion of the developer population that a given developer has collaborated with directly or indirectly up to a given point in time. The ratio of direct to indirect collaborations ($COL.DIR.*$) indicates the proportion of activities a developer performs on artifacts that have been previously changed by developers that the developer is familiar with. All dispositional characteristics are qualified by a type of artifact to which they relate to. That is, a developer d may have collaborated with a set of other developers D^p on a project p, but the developer may have collaborated with only a subset $D^f \subseteq D^p$ of these developers on a particular file artifact $f \in A^p$ from the project p, and it may be even a further subset $D^m \subseteq D^f$ of these for artifacts at a finer level of granularity, such as a method $m \in A^f$ where $A^g = \forall a^{g+1} : contains(a)$.

4.2.6. Aggregation and Distribution

Given that measurements and observations are often made at different levels, both spatially and temporally, the aggregation and distribution of the characteristics needs to be considered.

4.2.6.1. Spatial

From a spatial perspective, we are concerned with structural levels of artifacts, such as methods, classes, files, packages, components, projects. Typically there are containment relationships between these different structural levels, where classes contain methods, files contain classes, etc. Measurements at finer levels need to be aggregated into coarser levels of granularity where applicable. For relative measurements, we aggregate the measurements by computing the mean of the values based on the containment relationships. For absolute measurements we aggregate the measurements by computing both the mean and the sum of the values based on the containment relationships.

Inversely, measurements at coarser levels need to be distributed into finer levels of granularity where applicable. Here we rely on an approach that is somewhat similar to the one discussed in Section 3.5. Since we focus on the distribution of measurements regardless of factors, we only weight them against the size of a contained artifact or distribute them evenly.

4.2.6.2. Temporal

The frequency of available measurements and their intended application determines the temporal granularity. When it comes to developer states, and project phases, measurements from individual activities need to be aggregated. As discussed in the individual dimensions above, we consider the mean and standard deviation for situational characteristics up to the state of a developer as part of the dispositional characteristics. While in this case we considered the cumulative distributional characteristics, each state is also characterised by the distributional characteristics for the situational characteristics related to the activities performed by the developer in that state.

In order to determine the state boundaries, we consider two kinds of linear scales. The contribution-based linear scale increments the developer state after every n activities. The time-based linear scale increments the developer state every n time periods. In addition, we also consider a time-based non-linear scale where activities that are performed close to each other determine the developer state. Finally, we consider a clustering approach for identifying different modes of operation of a developer based on the contribution behaviour. In this approach, the developer may switch between the modes multiple times, resulting in a cyclic graph, rather than a linear sequence of states. Regardless of the approach, we need to aggregate measurements temporally to characterise the developer states.

4.3. Patterns and Applications

So far, we considered the characterisation of individual development activities with respect to the state of the artifacts on which they are performed and the state of the developers who performed them. Sequences and groups of development activities can be considered for determining the wider context of an activity. Additionally, the number of available characteristics grows considerably with the addition of new sources of information, levels of granularity, as well as depending on the applied aggregation and distribution strategies. By definition, each activity by a developer is characterised by a large number of values related to the state of the artifact before and after the activity, the state of the developer, as well as the activity itself. This results in vast amounts of data related to the behaviour of developers. In this section we discuss means for making sense of the collected data and identifying potential patterns, as well as applications for gaining further insights.

Software visualisation can be a a useful tool for navigating and comprehending vast amounts of data in order to gain initial insights from the available data. Various aspects of the available data can be highlighted and put into relevant contexts for visual inspection. The obtained qualitative insights can serve as the foundation for automated assessment by means of data mining and simulation, which in turn produce quantitative insights. The results from the automated assessment can then be visualised as well for further inspection.

4.3.1. The Importance of Characteristics

High-dimensional data can pose some challenges for its subsequent use in various applications such as visualisation and data mining. As one of the main goals of this thesis is identifying the circumstances under which developers are likely to cause events of interest, thus possibly contributing to technical risks, we want to focus on the characteristics that are associated with causing events of interest. Directed data mining approaches often rely on various measures of the importance of individual characteristics with regard to the target characteristic. One such measure is *information gain* [124] which serves as an indication of the ability of a characteristic to partition a data set with regard to the target characteristic. In order words, it provides means to measure the information regarding the values of the target characteristic provided by the values of a given characteristic. Information gain is used as the basis for machine learning with decision-trees. It is also used for attribute selection in order to reduce the number of characteristics to a subset of the most important ones with regard to the target characteristic. Information gain has certain shortcomings with respect to characteristics with many values. Other measures for the importance of characteristics, such as *gain ratio* [124], seek to address these shortcomings. The gain ratio provides means to penalise characteristics with many uniformly distributed values. However, the gain ratio has some shortcomings of its own, as do other related measures. In the following we will rely on information gain as basis for the examples. In practice, other measures for determining the importance of the different characteristics can be used as well.

When used in attribute selection or machine learning, information gain is typically calculated over the all the available data (which is usually the training data). This provides a single score for each characteristic. This score can be used to order the characteristics based on their importance resulting in a ranking of characteristics. In the context of this thesis, this can be useful for describing various partitions such as data related to a particular developer or a group of developers. As such, it can also be useful as an indication of similarity between individual developers or groups of developers. The ranking of characteristics in high-dimensional data can be challenging to represent visually in a useful way. *Kiviat diagrams* (also known as spider charts, star plots, radar charts, among other terms) [27] can be helpful for displaying high-dimensional multivariate data. The scale for each characteristic is plotted as a radial line (or a ray) from the center of a circle to its perimeter boundary. The radial lines are generally arranged at equal angles between any two lines. The value of each characteristic is plotted as a point on the corresponding radial line. A polygon defined by connecting the points corresponding to each value defines a shape or signature of data being represented. For comparison purposes, it is necessary to align the rays and scales in a consistent manner. While it may be difficult to compare the exact positions of the plotted values visually, patterns emerging from notable differences and similarities between different data sets can be easily recognized. The dominant characteristics of different data sets can be accessibly inspected and compared in this way.

An example of a Kiviat diagram for the ranking of the different characteristics for developer A at source code file level of granularity from the *log4j* project is shown in Figure 4.6. The names of the different characteristics are included for reference, however, in subsequent examples they will be omitted for brevity. In this example, we can observe that certain spatial characteristics are ranked rather high, whereas most collaboration characteristics are ranked very low. Overall, the characteristics are well differentiated with regard to their rank.

An example for comparison based on the visual representation of multiple rankings is shown in Figure 4.7. In this example we consider two developers — A and B, which have rather different rankings. In addition, we compare the rankings for both developers with the rankings over the first half of the available data. In a practical scenario where the available data is used for data mining, the first half of it may be used as training data (depending on the chosen partitioning and evaluation approaches). When comparing the rankings for the first half with the rankings for the complete data set, we observe that while the rankings for developer B are rather similar, the rankings for developer A show some differences. For illustrative purposes, we only consider a visualisation of the ranking for different developers. The same approach can also be applied for studying and comparing rankings at other levels of granularity, as well as for other partitions, such as whole projects, individual artifacts, groups of developers, or also groups of activities for a developer.

The ranking can also be used to determine how the importance of the individual characteristics evolved over time. By applying more generalized approach for contemplating the importance of individual characteristics at different points in time, we can assess the stability of the different characteristics [41, 86, 177], and also infer potential changes in behaviour.

Consider the example in Figure 4.8. It shows the evolution of the ranking of the characteristics for developer A from the *log4j* project with regard to causing events of interest of type *BugFix*. The ranking based on all available data determines the final placement of the characteristics which is plotted on the y-axis. The available data is split in n increments plotted on the x-axis (in this case $n = 10$), where starting from all data,

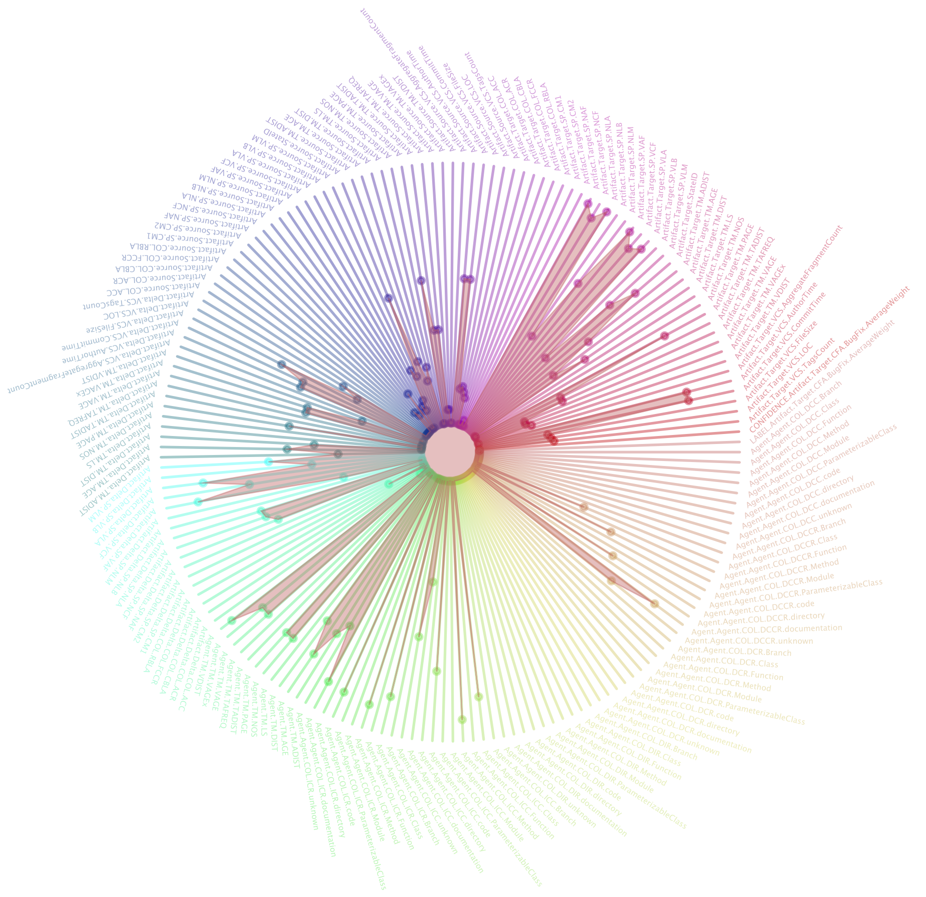
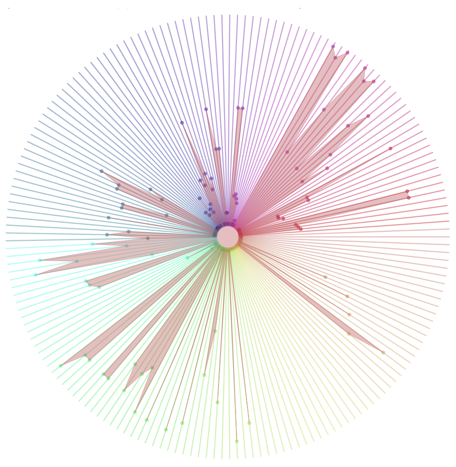
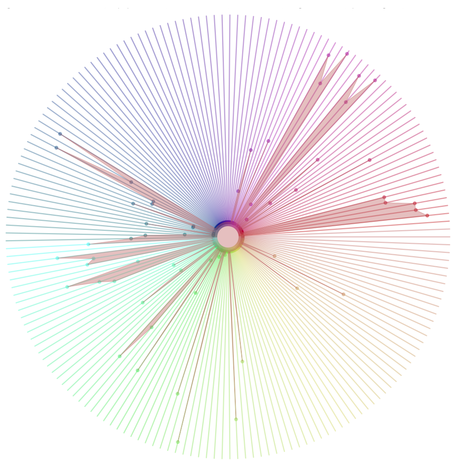
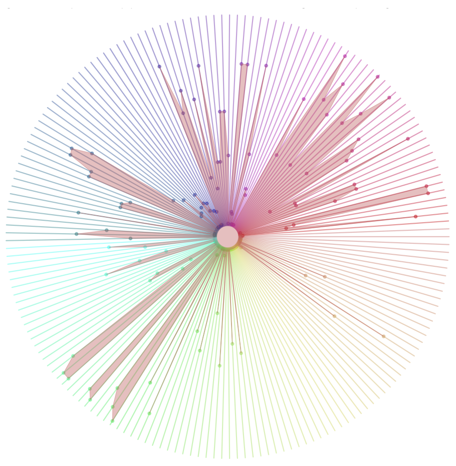


Figure 4.6.: Characteristics ranking for developer *A* in log4j (see Figure A.6–A.7 for a larger version)

one increment is removed and ranking is performed again for the $n-1$ increments. Then the next increment is removed and ranking is performed again for the $n-2$ increments, and so on. Characteristics that have the same score are placed at the same rank (hence thicker lines, especially at the lowest ranks). Characteristics that have a score of 0 in all the available data are discarded. The legend includes the highest ranked characteristics (ordered by rank). In this example, we can observe that while the highest ranks are rather stable, there is some variance in the lower ranks especially around the middle. Depending on the number of activities by a developer, a larger number of increments may provide a more detailed view on the dynamics in the ranking of the characteristics, whereas a lower number of increments may mask some peculiarities. Figure 4.9 shows the ranking evolution for the same developer at a higher resolution with $n = 50$. In this


(a) Developer *A*


(b) Developer *A* (first half)


(c) Developer *B*

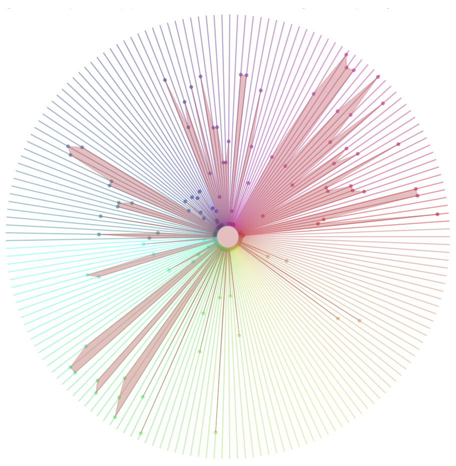

(d) Developer *B* (first half)

Figure 4.7.: Comparison of characteristics ranking for two developers in log4j (see Figure A.8–A.11 for a larger version)

case we can observe somewhat increasing stability in the last 20% of activities

Comparing four different developers as shown in Figure 4.10 can yield some further insights based on visual inspection. Somewhat similar to last 20% of developer *A*, the ranking of the characteristics for developer *B* is mostly stable after the first 20%. On the other hand, the ranking evolution for developers *C* and *D* appears exhibit more drastic changes where ranks may vary substantially at the different increments.

So far we considered data from the start up to a certain number of increments. This mimics practical application, where as more data becomes available during the lifetime of a software project, more insights can be gained with regard to the importance of the individual characteristics. In some cases, there may be very substantial and permanent changes in the importance of the characteristics after a certain point, due to a shift to new technology for example. In such cases, it may make sense to only contemplate distinct periods in the lifetime of the project in isolation. This can be based either on considering different increments in isolation, e.g. only the data between $n-1$ and $n-2$, or by using global offsets and limits. While we only considered examples for developers, the evolution of the importance of the characteristics can be studied at the project level, for individual artifacts, as well as for groups of developers, e.g. small developers.

As noted by Kalousis et al. [86], the ranking of characteristics can be indicative of their importance in data mining. However, it cannot serve as a definitive estimate of the discriminatory power of the characteristics, since it does not construct classification models whose error could be estimated. Conversely, instability is not necessarily associated with low classification performance either, as noted in [86]. Stable ranking should intuitively lead to stable classification models, periods of stability can be potentially useful for data partitioning and filtering.

4.3.2. Mapping Developer Activities in Time

Developers are usually performing activities on different artifacts at different points in time. Sometimes a single developer may perform a burst of activities over a short period of time, spanning one or multiple artifacts. At other times, several developers may collaborate on a group of artifacts over a longer period of time. A visual inspection such as the one proposed by Girba et al. [57] can be helpful for gaining an initial insight into how the activities of developers are related to artifacts over time.

Figure 4.11 shows an example for a projection of all activities on source code files for the *log4j* project, which will be referred to as an artifact activity map. Each artifact is assigned a horizontal "lane" on the vertical axis. The artifacts may be ordered by time of creation or based on other characteristics. All activities for an artifact are plotted as squares on the same horizontal line corresponding to the artifact. The location of the square on the horizontal axis corresponds to the time between the recorded start and end of the project. Activities on multiple artifacts at the same time are plotted under

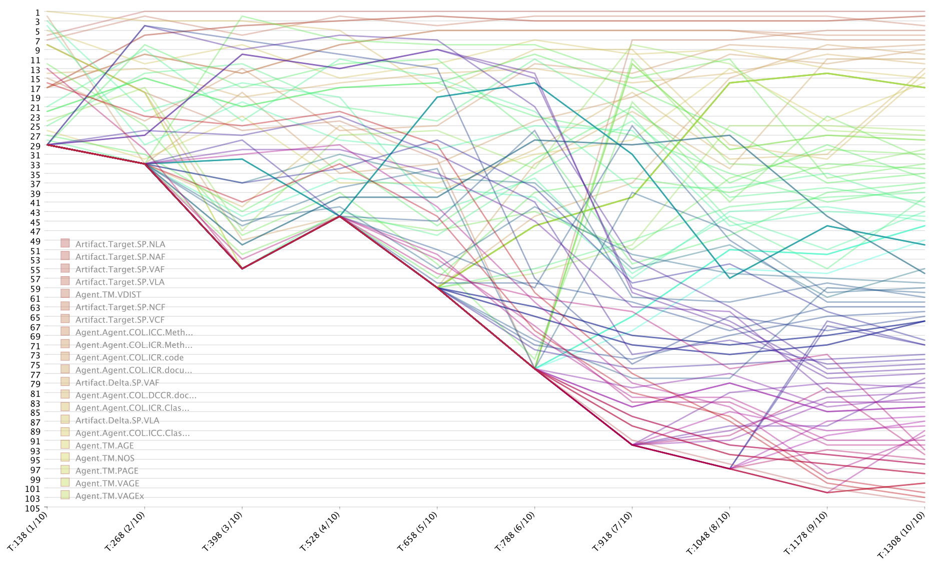


Figure 4.8.: Characteristics ranking for developer *A* in log4j over time (see Figure A.12 for a larger version)

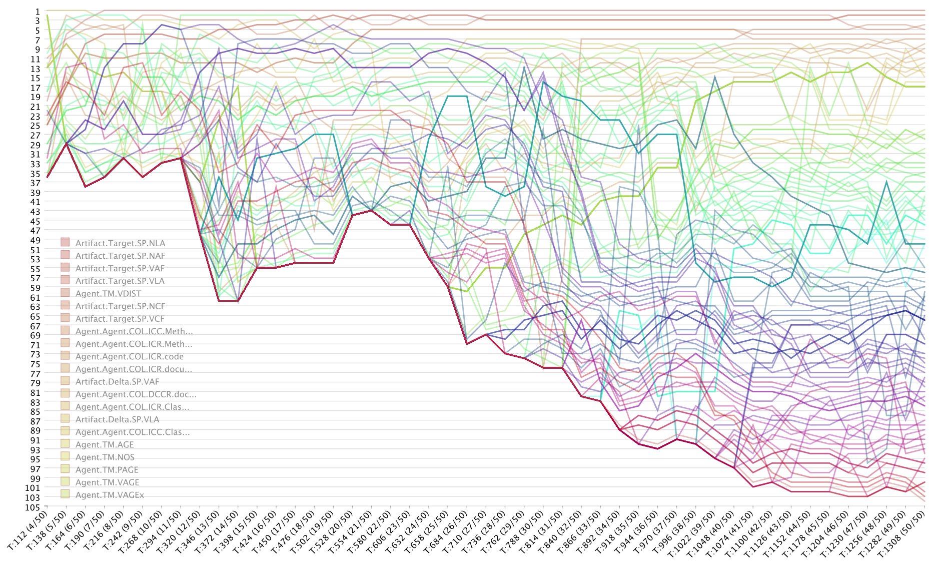
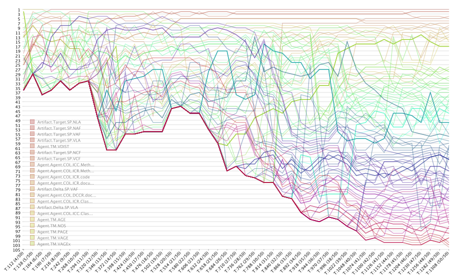
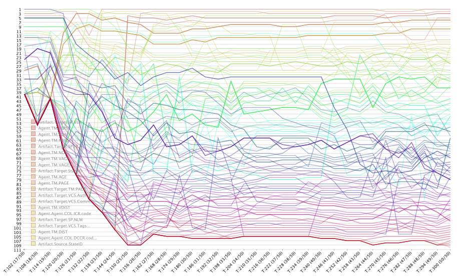
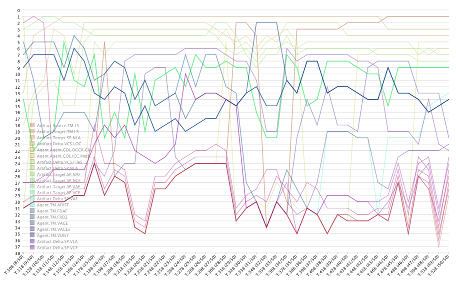


Figure 4.9.: Detailed characteristics ranking for developer *A* in log4j (see Figure A.13 for a larger version)


(a) Developer *A*


(b) Developer *B*


(c) Developer *C*

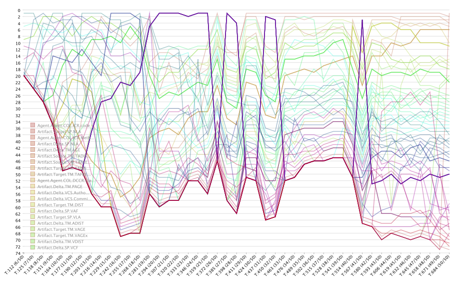

(d) Developer *D*

Figure 4.10.: Detailed characteristics ranking over time for different developers in log4j (see Figure A.13–A.16 for a larger version)

one another. The color of the squares corresponds to the developer that performed the activity. The narrow green bands near the end of the recorded period indicate that the green developer performed many activities on many of the artifacts over a short period of time or at the same time. Similar occurrences can be observed for the dark blue developer around the middle of the recorded period as well as for the light green developer at various points in time. Most of the activities of the blue developer on the other hand are performed around the middle of the recorded period, on artifacts created around the same period. In the last third of the recorded period there were generally fewer activities than during the first half of the recorded period (except for the above mentioned narrow bands), performed by a small number of developers, potentially indicating a certain level of maturity of the project or perhaps also a loss of interest from developers.

This visual representation can serve as the basis for displaying additional information as overlays. One such overlay is shown in Figure 4.12, where the likelihood of an activity for causing an event of interest of the *BugFix* type is shown as red circles on top of the squares for each activity. The size of a circle is proportional to the likelihood for causing an event of interest. In this example, we can observe that while many activities

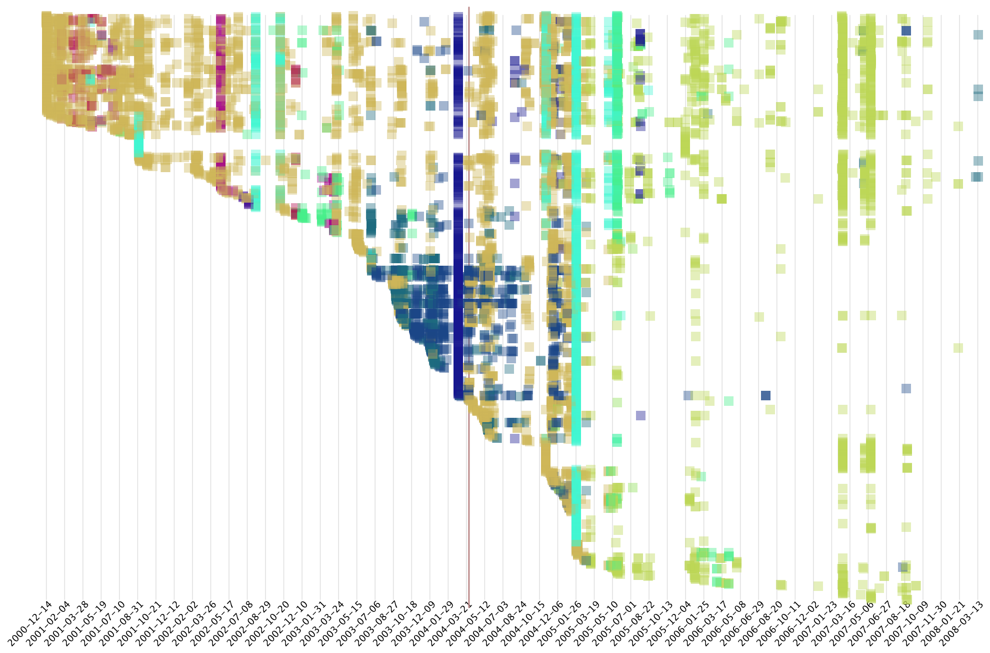


Figure 4.11.: Developer activities on source code files for log4j

have a risk for causing an event of interest, in a lot of the cases the likelihood is rather low. Most of the high-likelihood cases occur in the second half of the recorded period.

4.3.3. Roles and Ranks

Over the course of a project, developers usually assume different roles, either explicitly or implicitly, which in turn are associated with different responsibilities and potentially also result in different behaviour. Whether the role determines the behaviour or whether the behaviour determines the role can be considered a chicken-and-egg problem. It can be assumed that when roles are assigned explicitly, the role is expected to determine the behaviour, whereas implicit role assignment is based on the experience of a developer indicated by previous behaviour. Even with explicit role assignment, the decisions are typically based on previous experience for which there may be no detailed behaviour records for a particular project, e.g. when the experience was acquired within a different project or organisation. Whichever the case, the different roles are expected to have different influence on the development and also on the community in the broader context (e.g. answering questions on mailing lists, etc.).

We consider means for the identification of different roles and the assignment of these roles to developers based on observed behaviour (implicit role assignment). Thus, the

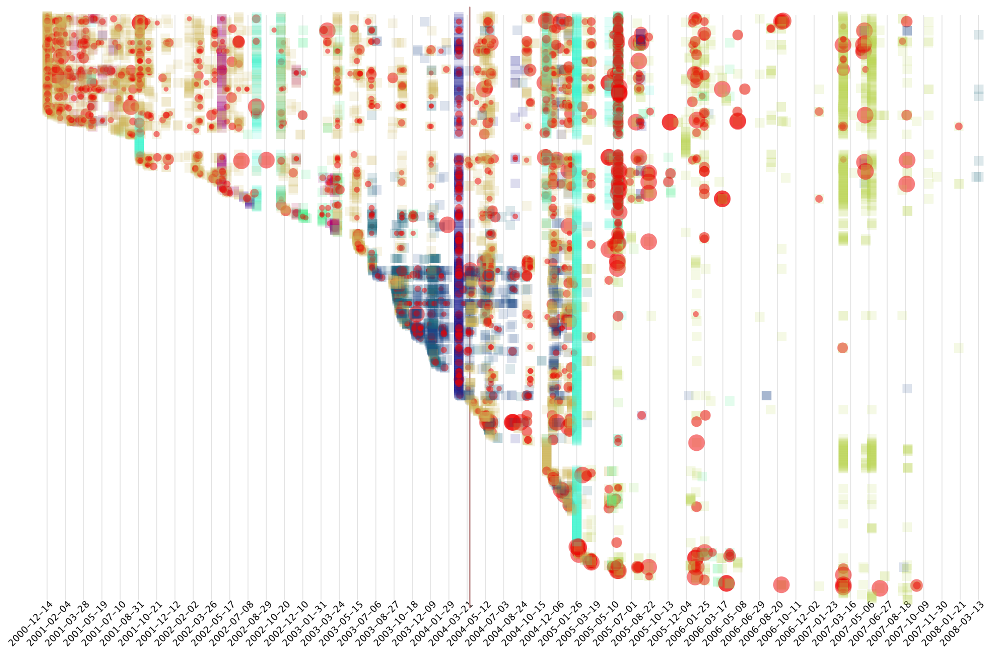


Figure 4.12.: Developer activities on source code files for log4j with risk overlay

role of a developer at a given point in time is determined based on the overall behaviour of the developer up to that point in time. We consider different perspectives on developer roles, partially based on the previous approaches discussed in the literature. One perspective is to consider the role of a developer as a dynamic characteristic of the behaviour of the developer at a given point in time. As such, the role of a developer may change over time, depending on their own contributions as well as on the contributions of other developers working on a project. For example, a core developer that becomes inactive for a period of time would eventually move down the ranks as other developers contribute more and more. In this regard, rather than having descriptive roles, such as core and peripheral developers, we can pursue a dynamic rank-based approach.

The ranking approach is inspired by Wald's approach for graphical sequential analysis [182]. It plots the path of each developer according to a measure of the developer's own contributions against the contributions of other developers in two-dimensional (x,y) space. Contributions may be measured based on the number of activities a developer has performed, the size of the changes within the activities, the amount of code owned by the developer, or other measures. Consider the artificial example shown in Figure 4.13. A developer's own contributions are plotted on the vertical (y) axis, whereas the cumulative contributions of other developers (foreign contributions) are plotted on the horizontal (x) axis. In each state, a developer moves up according to the size of the contributions, and right according to the size of the contributions of all other developers at the point in time associated with the developer's state. In this example, the red developer made an initial contribution of size 10, thus at time point t, the red developer is located at $(0,10)$, represented as a red circle. At the same time, the yellow developer has not yet made any contributions, therefore, the yellow developer is located at $(10,0)$. Since the yellow developer is not active at the given point in time, the location is not marked by a circle (implied location). The blue dashed line represents the project front at a given point in time. The position of a developer on project front line represents the rank of the developer at the corresponding point in time. At time point t, the red developer has a higher rank than the yellow developer. The next contribution of size 5 at time point $t+1$ is by the yellow developer, therefore, the yellow developer moves to $(10,5)$. As the red developer is inactive at this point in time, the implied location for the red developer is $(5,10)$. At this point in time the red developer still has a higher rank thank the yellow developer, although the gap is narrower. The next contribution of size 5 at time point $t+2$ is again by the red developer, both developers are relocated accordingly as the ranking remains and the gap widens again. The last two contributions are by the yellow developer. The first of size 5 at time point $t+3$ narrows the ranking gap again, and the second of size 10 at time point $t+4$ places the yellow developer at a higher rank than the red developer, which, due to inactivity, has been demoted to a lower rank. This is a simplified example serving as an illustration of the basic principles for constructing the graphical representation of the path of a developer through the ranks at different points in time in relation to other developers based on the contributions of each developer against the contributions of all other developers. In practice multiple developers make contributions of varying sizes, often in quick succession, but sometimes also only sporadically.

An example for a real-world project is shown in Figure 4.14. In addition to the elements discussed so far, in this case, black lines originating from the lower left corner represent various thresholds for different ratios between own and foreign contributions. The example is based on the number of activities at the file level of granularity where only activities on source code files are considered. In addition, only the project front in the last recorded point in time is shown as a blue solid line. In this example, we can observe that the yellow developer was dominant from the very start of the project, but became rather inactive towards the end of the recorded period. With regard to the subsequent ranks, there were several changes where the dark green and dark blue developers made some considerable contributions to rise through the ranks, followed by the light blue developer making larger contributions over rather few states, with some inactivity in between. Both of the latter were overtaken by the light green developer who rose through the ranks in a rather short time towards the end of the recorded period.

Figure 4.14 highlights the relationships between own and foreign contributions among the individual developers over time. To further emphasise the ranking changes

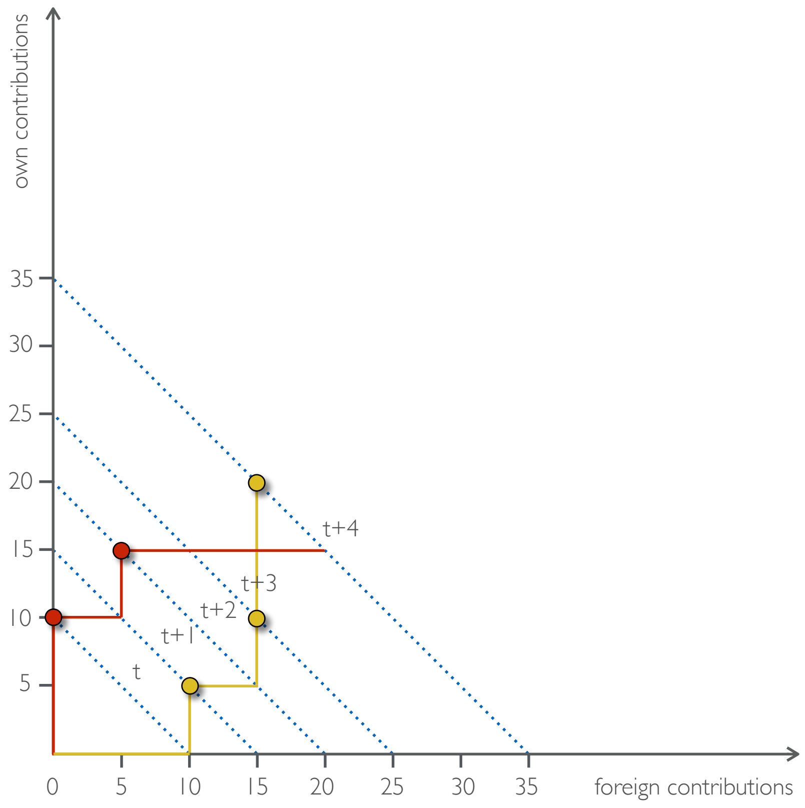


Figure 4.13.: Developer ranking with graphical sequential analysis

over time, we can project the ranking on the project front over discrete time (incremented with each developer state), as shown in Figure 4.15. In this projection, the ranks of the developers are plotted on the vertical axis and the developer states are plotted incrementally on the horizontal axis with a coloured circle at the corresponding rank. A line in the corresponding colour connects the states of a developer during inactivity. Developers that cannot be differentiated due to them being at the same rank given point in time are displayed with overlaid lines. For example, in the beginning of the project when only the yellow developer is active, all other developers are sharing the lowest (second) rank. Over time, the lowest rank becomes more and more differentiated.

With this approach, there is a natural penalty for late joiners by default. As a consequence, they need to make more contributions in order to go up the ranks. As observed in Figure 4.14, it is still plausible with very active contributors where at the same time high-ranked developers become inactive for a period of time. The penalty can be further offset by using a gradual descent or by using a moving window strategy where only contributions from a limited period are considered so there can be a yearly rank,

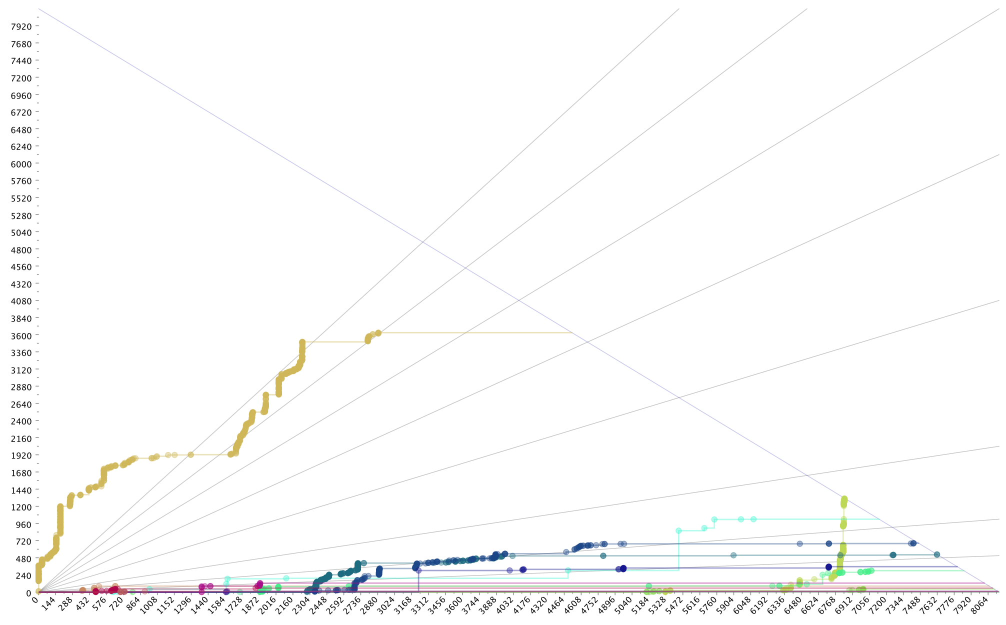


Figure 4.14.: Developer ranking for log4j

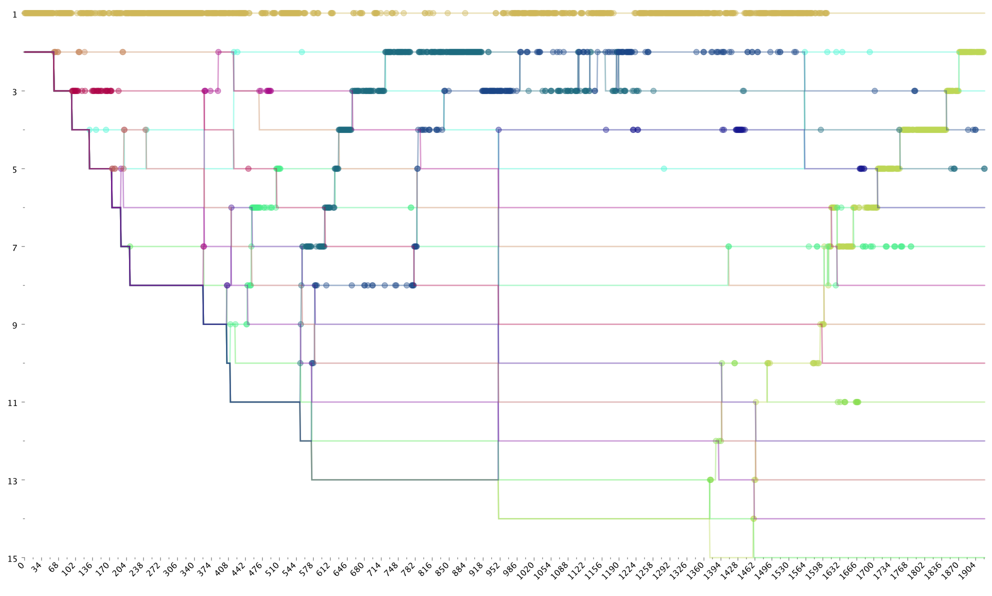


Figure 4.15.: Developer ranking at project front for log4j

monthly rank, and global rank or rank over last *n* states or contributions. Temporally constrained ranks are related to the operational mode of a developer, whereas the global rank is indicative of the role.

Based on the visual inspection approach, more sophisticated approaches for ranking and role assignment, as well as related characteristics can be derived from the underlying representation. So far we considered only relative ranking, that is the relative ordering of the developers on the project front. As indicated in Figure 4.14, different thresholds may be used to partition the space into ranks and roles. These may be static — based on fixed ratios between own and foreign contributions (as shown in Figure 4.14), or dynamic — based on ratios derived from the number of involved developers or also on the positions on the project front. Characteristics related to the time spent at a rank or the number of rank changes can provide further insights into the dynamics of the roles developers play within a project.

So far, the approach has only provided insights into the dynamics of the contribution behaviour of the developers according to a certain measure of contributions, such as number of activities, size of contributions, size of owned artifacts, etc. We are ultimately interested in the circumstances and consequences associated with the observed roles and ranks, as well as potential patterns related to these. For this purpose, we can project various characteristics on the resulting visual representations. For example, we can project the estimated risk of each developer state. The risk of a developer state can be derived from the risks associated with each activity performed at the corresponding developer state. The risks associated with an activity are based on the likelihood of the activity causing an event of interest, such as a fix. The resulting projections derived from Figure 4.14 and Figure 4.15 are shown in Figure 4.16 and Figure 4.17, respectively. The risky developer states are highlighted with red circles, where the size of the circle is proportional to the amount of risk. Based on the resulting visual representations, we can observe that in this example the most active developers are also most likely to be in a risky state, especially at high ranks, but also when rising through the ranks. Lower ranked developers in this example are generally less likely to be in a risky state.

4.3.4. Identifying Similar Activities

Developers may often perform similar activities throughout the lifetime of a project. Determining whether activities can be considered similar can be based on the intent of the activity or on the observed circumstances determined by measurable characteristics of the activity as well as of the context in which it was performed. In this section, we are concerned with the latter. We consider undirected data mining approaches, such as clustering, as means to identify groups of similar activities. Such groups can be used to determine the mode of operation of a developer or even a whole project. Applying a variant of the *k-means* clustering algorithm [5] over all activities for a developer or a project can provide a partitioning of the activities based on similarity. The parti-

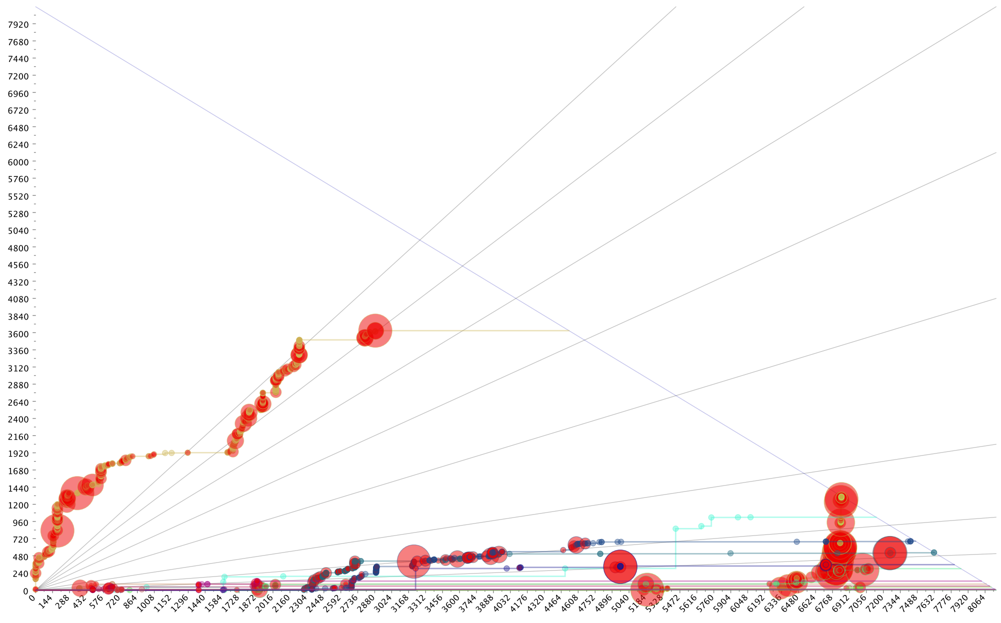


Figure 4.16.: Developer ranking for log4j with risk overlay

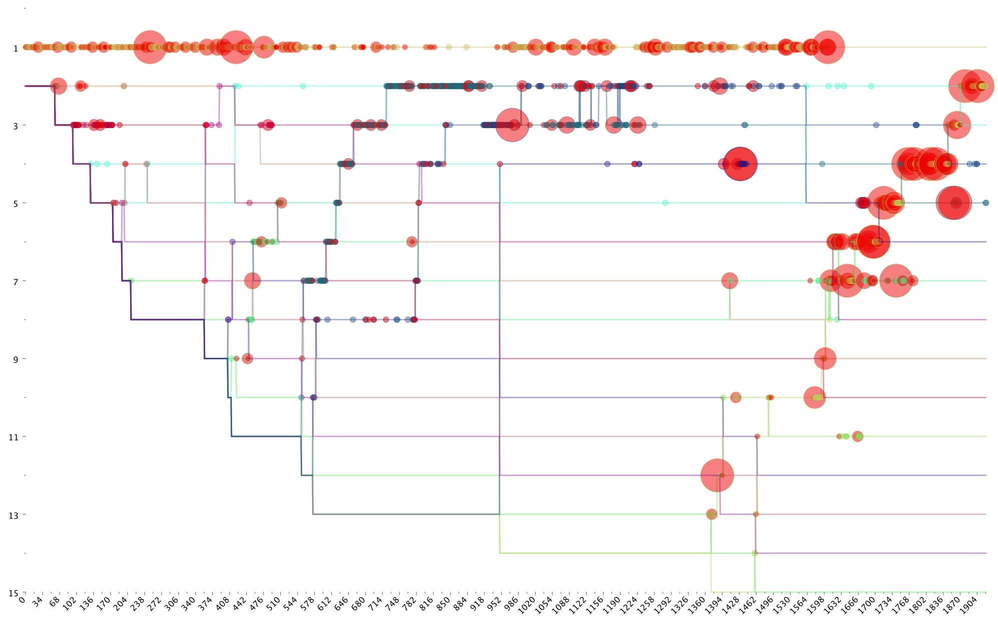


Figure 4.17.: Developer ranking at project front for log4j with risk overlay

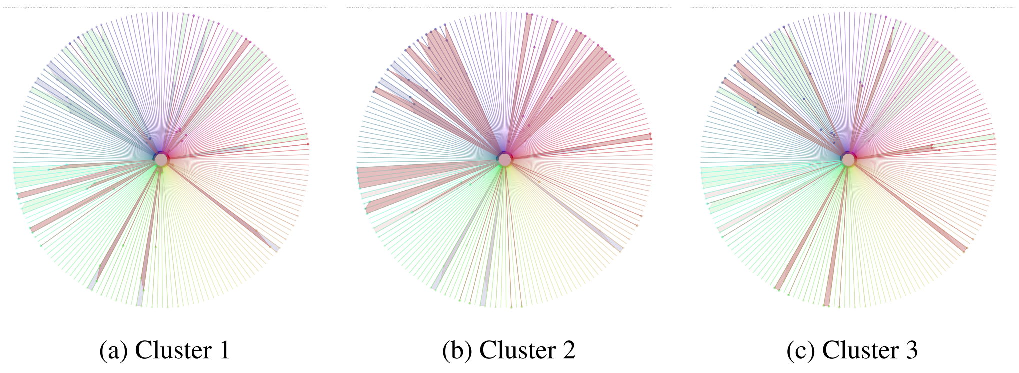
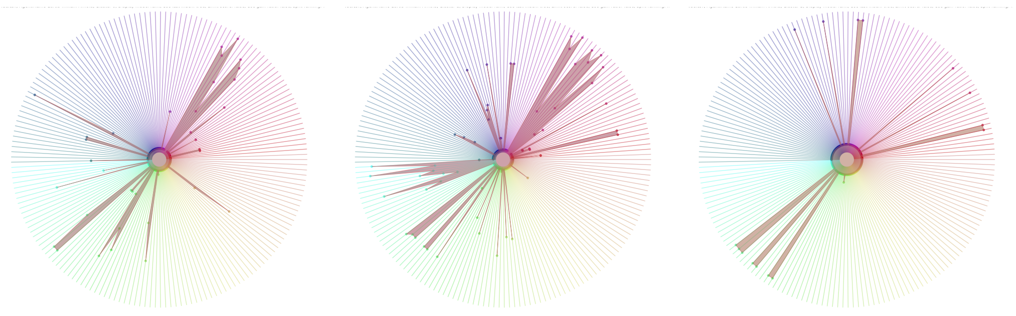

(a) Cluster 1 (b) Cluster 2 (c) Cluster 3

Figure 4.18.: Defining characteristics for developer *A* in log4j across three clusters (see Figure A.17–A.19 for a larger version)

tioning can be helpful for more refined assessment providing additional capabilities for visual inspection and data mining. Predictive models tailored for individual groups can be more specific and new activities can be evaluated against the models for the similar groups, rather than a generic global model. Additionally, there may be transfer opportunities between similar groups across developers and projects. Similar across developers and project groups can be identified based on meta-clustering using the resulting clusters for individual developers and whole projects as input

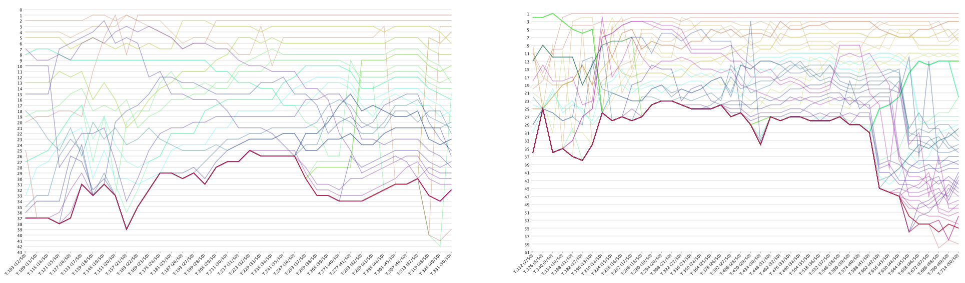
Depending on the characteristics of each cluster, a corresponding description can be added. Similar to Sudau et al. [171], we use Kiviat diagrams to visualise the defining characteristics for each cluster as a visual signature and also for further visual inspection. Consider the examples for developer *A* from the *log4j* project shown in Figure 4.18. Clustering the activities of the developer across three groups ($k = 3$), the normalised values for the cluster centroids are plotted on the radial lines for each characteristic. The defining characteristics for each cluster are highlighted, with additional overlays in the background indicating the defining characteristics of the other clusters for comparison. We can notice that there is some overlap between the defining characteristics for cluster 1 and 3, however, there are also some differences. In contrast, cluster 2 is more visibly differentiated from the other clusters.

Another way to characterise different groups is by the importance of the characteristics with regard to causing events of interest, as discussed in Section 4.3.1. Similar to the visualisation of the overall ranking of characteristics, we can also inspect the ranking of characteristics in each cluster. An example for the clusters discussed above is shown in Figure 4.19. In this case we can observe that while there is some overlap between clusters 1 and 2, cluster 3 is more differentiated from the other clusters with respect to the ranking of characteristics. At the same time, the ranks of the characteristics are not very well differentiated in cluster 3. If we consider how the ranking of



(a) Cluster 1 (b) Cluster 2 (c) Cluster 3

Figure 4.19.: Ranking of characteristics for developer *A* in log4j across three clusters (see Figure A.20–A.22 for a larger version)



(a) Cluster 1 (b) Cluster 2

Figure 4.20.: Characteristics ranking over time for developer *A* in log4j (see Figure A.23–A.24 for a larger version)

the characteristics changed over time, we can gain further insights into the stability of the rankings. Consider the comparison between clusters 1 and 2 shown in Figure 4.20. We can note that in cluster 1 steady variation with characteristics slowly going up and down the ranks. In contrast, in cluster 2 there were three distinct periods, where there was high variation in the beginning and in the end with a period of relative stability in the middle. Different views on the distribution and importance of characteristics can serve complementary roles in describing and comparing the individual groups.

Clustering can also be applied on part of the activities in order to determine initial groups, where subsequent activities can then be assigned to the initial groups depending on which group is most similar to each activity. This mimics a practical application scenario, where clusters may be identified based on the available data at a given point in time, and subsequent activities are assigned to the most similar group. This can be combined with the artifact activity map in order to obtain an insight into the location of the activities in each group across artifacts and across time. Figure 4.21 shows an

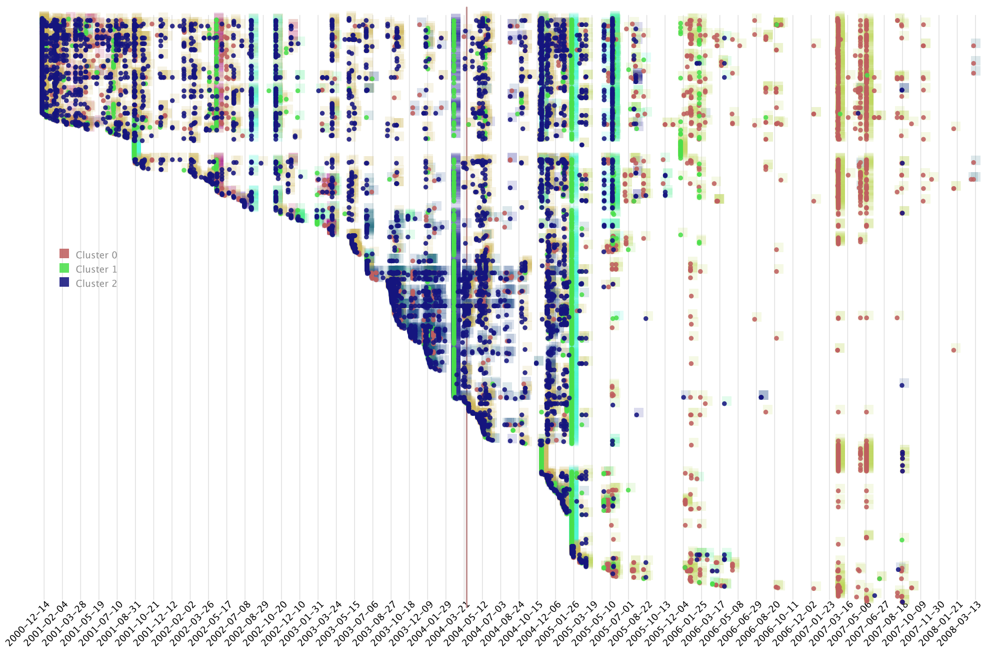
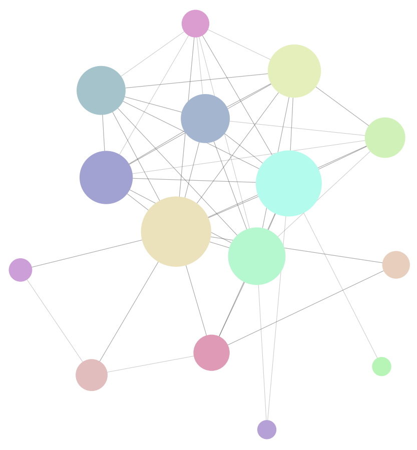


Figure 4.21.: Developer activities on source code files for log4j with cluster overlay

example for a projection of cluster assignments as an overlay on top of the artifact activity map for the *log4j* project. The little circles in the lower left corner of each square indicate the cluster to which an activity was assigned. In this example we consider all of the activities on source code files for the project which are clustered across three groups ($k = 3$). Alternatively, we can also inspect only the activities of a single developer. We can observe that in the first half of the recorded period, most activities were from the blue cluster, with some occasional activities from the red and green clusters. Towards the end most activities were from the red cluster with occasional activities from the green and blue clusters. The activities from the green cluster are mostly concentrated in two narrow bands towards the middle of the recorded period. Finally, the red line near the middle of the recorded period is the boundary for the initial clustering.

4.3.5. Collaboration

In Section 4.2.5 we discussed different notions related to collaboration. By using collaboration relationships, we can construct collaboration graphs for direct and indirect collaborations. A visual representation of collaboration relationships can be helpful in understanding the position of a developer with regard to other developers as well as distribution of developers with regard to collaboration. Graph visualisation techniques


(a) Direct

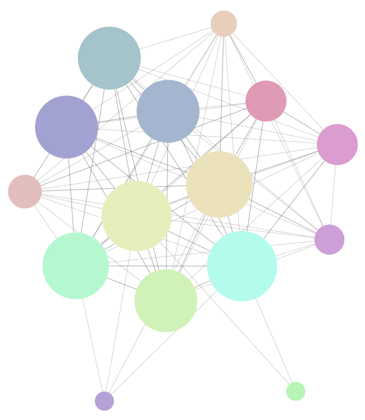
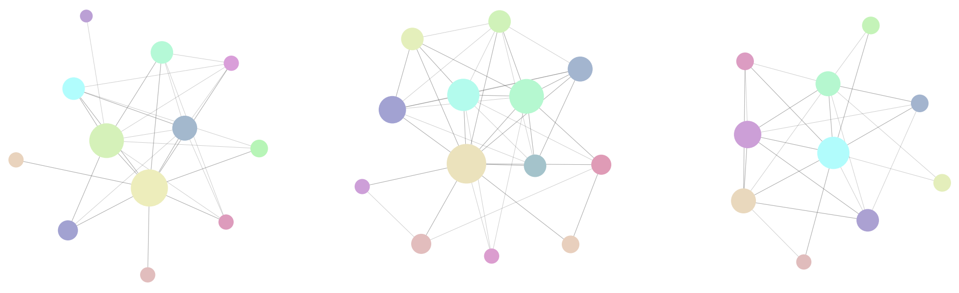

(b) Indirect

Figure 4.22.: Collaboration between developers in log4j

and various layout algorithms are frequently used to highlight different characteristics of graphs for visual inspection. Force-directed graph layout algorithms in particular aim to position nodes of a graph in two or three dimensional space so that the result is suitable for visual interpretation, while focusing on a particular aspect of the graph. We use the *Fruchterman-Reingold* algorithm [52] which produces compact graphical representations placing highly connected nodes (representing individual developers) closer together, typically towards the core of a circle, while less-connected developers are placed on the periphery. The size of a node can be made proportional to the number of connections of the node. An example for collaboration at the source code file level from the *log4j* project is shown in Figure 4.22. We can observe that with regard to direct collaboration, there are several less-connected developers in the periphery. In terms of indirect collaboration, however, most developers are much more highly-connected, with only two developers having a lower number of indirect collaborations.

Collaboration visualisation can also be applied to individual clusters of similar activities, providing further means to characterise the cluster. Consider the examples for direct collaboration within the three clusters from the *log4j* project shown in Figure 4.23. In cluster 1 we can note that two developers are more central, whereas in cluster 2 three developers are more central, and in cluster 3 only one developer is more central. We note that not all developers are represented in each cluster.

Distinct highly-connected groups of nodes within a graph (connected components) may represent distinct closely integrated teams within a project. With force-directed



(a) Cluster 1 (b) Cluster 2 (c) Cluster 3

Figure 4.23.: Direct collaboration between developers in log4j within three clusters

algorithms such as the one used in the visualisations so far, these can be easily recognised upon visual inspection. Examples from two simulations are shown in Figure 4.24. The examples depict simulations for two and four teams, respectively. The developers are collaborating more with developers from the same team than with developers from other teams. Developers from the periphery are occasionally collaborating with developers from the different teams or among themselves. The identification of teams can be useful for further assessment, such as characterising a team as a whole, rather than individual developers.

So far we discussed visualisation of collaboration based on the notions described in Section 4.1.2. Further collaborative aspects can be considered as well. Utilising the cause-fix relationships between states of artifacts described in Chapter 3 we can also establish cause-fix relationships between developers across the different factors and use these for visualisation and inspection in order to gain further insights into the collaboration behaviour of developers.

By utilising the artifact activity maps and the developer ranking, we can also study potential collaboration patterns based on the behaviour of developers over time. Girba et al. [57] discuss a comprehensive approach for identifying behavioural and collaboration patterns based on ownership maps. They showcase ten patterns, three of which focus on collaboration, including *monologue* where most artifacts and most changes on them are performed by the same developer, *dialogue* where multiple authors perform changes during a given period and artifact ownership is distributed among them, with *teamwork* being a special case of *dialogue* where two developers perform a quick succession of changes over a short period of time. Another two patterns focus on the transfer of ownership over periods of time of different lengths. The *takeover* describes a case where a developer seizes ownership over large chunks of code in a short period of time. Similar to *takeover*, *familiarisation* also describes a change in ownership, but it takes

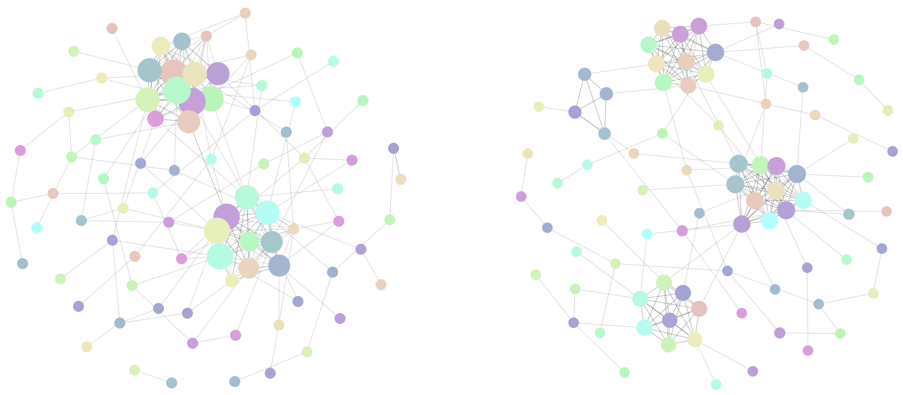

(a) 2 teams (b) 4 teams

Figure 4.24.: Collaboration simulations highlighting different teams

place over a longer period of time.

Based on the inspection and interpretation of visualisations, such as artifact activity maps, developer rankings, and ownership maps, an approach for the automated identification of such collaboration patterns can be defined by using the various collaboration related characteristics discussed in Section 4.2.5.

4.3.6. Predicting Causes for Events of Interest

Directed data mining techniques for classification and prediction are widely used in software engineering research for various assessment tasks [4, 65]. Change and artifact classification in particular has been actively researched as it can have direct practical consequences with respect to software quality assurance. Defect prediction as a specific application of directed data mining techniques for predicting which changes may introduce new defects or which artifact are most likely to contain defects has emerged as an area of research in its own right. The application of the directed data mining techniques results in a model of the relationships between a set of characteristics and a target characteristic.

With respect to characterising developer behaviour, directed data mining techniques provide means to model the relationship between the circumstances of a development activity and its outcome. The circumstances are defined by the characteristics discussed in this chapter. The outcome is defined by the consequences of the activity such as a high likelihood for causing an event of interest in the target state of an artifact resulting from the activity. The available data regarding the circumstances and outcomes of activities can be used to train a predictive model, which can then be used to evaluate evaluate future activities for which the outcomes are not known, based on the data regarding cir-

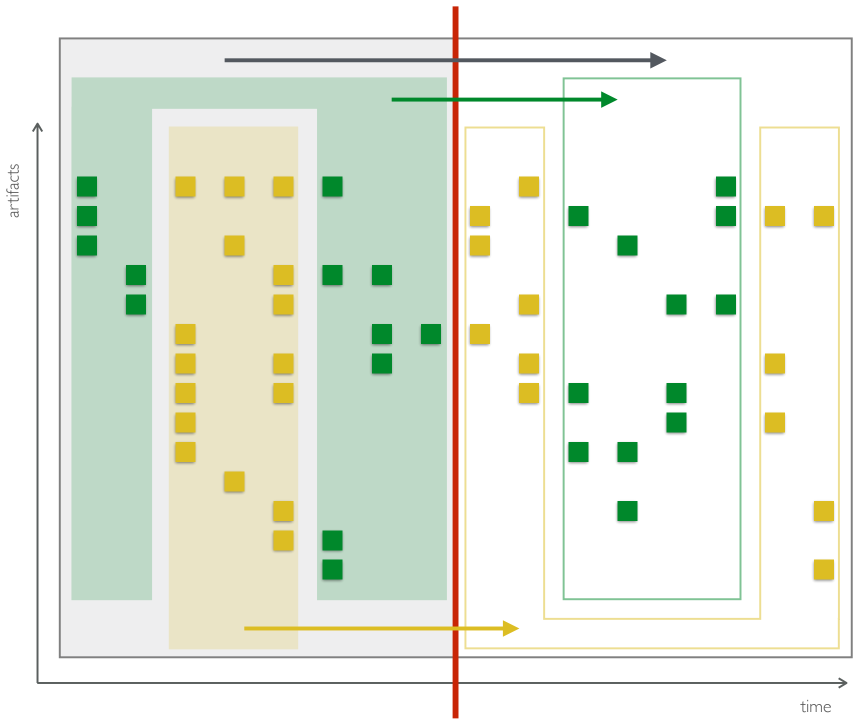


Figure 4.25.: Developer-centric and project-centric predictive modelling

cumstances alone. The behaviour of a developer with respect to causing a particular type of events of interest is then characterised by the resulting predictive model.

In the literature, predictive models are typically constructed for whole projects or even groups of projects, seeking to identify universal relationships between circumstances and outcomes and aiming for generic applicability. In contrast, we are interested in identifying relationships between circumstances and outcomes specific to each developer, emphasising the strengths and weaknesses of each developer. This can provide more refined feedback, tailored to a developer. To illustrate the difference, consider the conceptual overview depicting an abstract representation of artifact activity map shown in Figure 4.25. In project-centric predictive modelling (black arrow), all activities up to a given point in time (red vertical line) are considered as training data for predicting the outcome of all activities beyond that point in time. In contrast, in developer-centric predictive modelling (green and yellow arrows), for each developer, only the activities of that developer up to a given point in time are considered as training data. Future activities for each developer are then evaluated against the corresponding model for that developer in order to predict their outcomes. Each model reflects the specific circumstances associated with causes for events of interest for the corresponding developer.

As exemplified in Section 4.3.1, for each developer different characteristics may be determining the likelihood for causing an event of interest. Still, there may also be similarities between the behaviour of some developers, which can enable transferring models across different developers (commonly referred to as *transfer learning* [140] in the machine learning literature). Additionally, developers may be involved in multiple projects, which provides further transfer opportunities for models for the same developer (or also similar developers) across different projects. Similar approaches for transferring models across whole projects have been pursued in the literature in the areas of cross-project defect prediction [74, 198] and cross-company defect prediction [176]. A conceptual overview of the developer-centric predictive modelling and various transferring opportunities is shown in Figure 4.26. Given two projects *A* and *B*, where developers *pat* and *tom* are contributing to project *A*, developers *sue* and *ben* are contributing to project *B*, and developer *joe* is contributing to both projects, we define and exemplify the following modelling and transferring opportunities:

- using training data from project *A* for predicting the outcomes of future activities in project *A* (*within-project*)
- using training data from project *B* for predicting the outcomes of activities in project *A* (*cross-project*)
- using training data from developer *pat* in project *A* for predicting the outcomes of future activities of developer *pat* in project *A* (*same developer, within-project*)
- using training data from developer *ben* in project *B* for predicting the outcomes of activities of developer *sue* in project *B* (*different/similar developer, within-project*)
- using training data from developer *sue* in project *B* for predicting the outcomes of activities of developer *tom* in project *A* (*different/similar developer, cross-project*)
- using training data from developer *joe* in project *B* for predicting the outcomes of activities of developer *joe* in project *A* (*same developer, cross-project*)

Project-centric transfer opportunities (cross-project) for predictive modelling have been extensively studied in the literature [70, 74, 198]. Instead, we focus on developer-centric transfer opportunities, both within the same project and across different projects.

As indicated in Section 4.3.1, the importance of the characteristics determining the outcome of interest may change over time, which may also affect the stability and reliability of the predictive models over time. This may be due to new experiences of the developer, or also due to the dynamics of the overall circumstances in the project. Ekanayake et al. [47] found certain periods of variability and stability with respect to prediction performance for whole project over time. We are interested in the stability of predictive models for individual developers as well as for transferring models between different developers and different projects. Further inspection by utilising the different methods for visualisation and data mining can shed some light on the potential causes

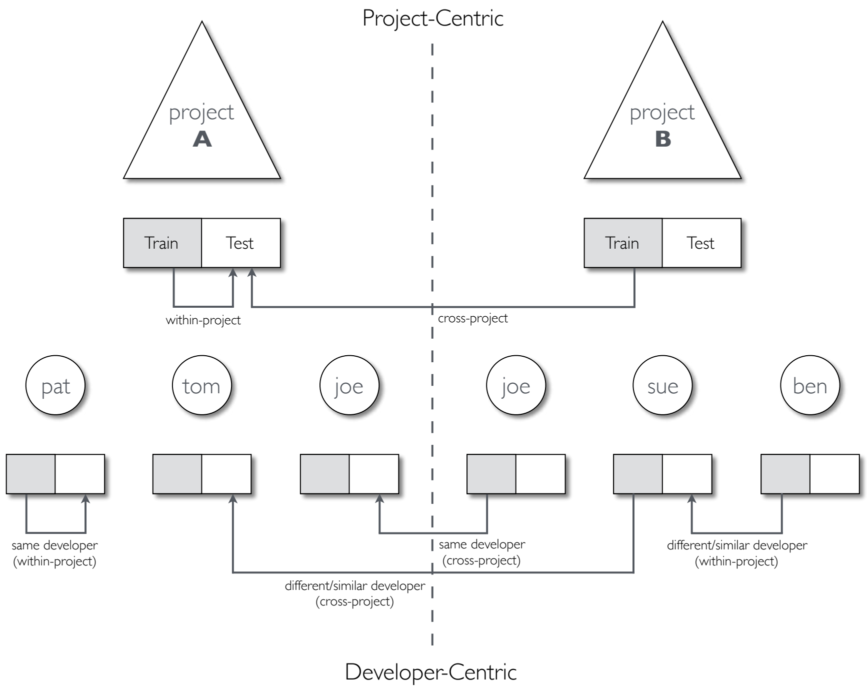

Project-Centric
project A
project B
Train
Test
Train
Test
within-project
cross-project
pat
tom
joe
joe
sue
ben
same developer (within-project)
same developer (cross-project)
different/similar developer (within-project)
different/similar developer (cross-project)
Developer-Centric

Figure 4.26.: Developer-centric predictive modelling and transferring opportunities

for shifts in behaviour and corresponding variability in prediction performance. Tan et al. [173] discussed challenges related to time-sensitive defect prediction, including imbalanced data and available knowledge at different points in time, representativeness of data from different periods of time, and aging of predictive models due to changes in experience, tasks, and styles of developers. They used resampling, gaps, and online learning to address these challenges. Kim et al. [95] noted that for classifying changes data for 100–200 changes is usually sufficient for training, with further data not contributing to significant improvements. In more general terms, this is related to data partitioning. With regard to the training and testing data for predictive modelling, we can introduce three gaps:

- *training offset* from the start of the recorded period to the start of the training data
- *testing offset* from the end of the training data and the start of the testing data
- *testing limit* from the end of the testing data to the end of the recorded period.

The different gaps are conceptually illustrated in Figure 4.27. By adjusting the size of the different gaps, we can explore various partitioning scenarios and schemes. We can then evaluate the stability of the predictive models across these scenarios and schemes.

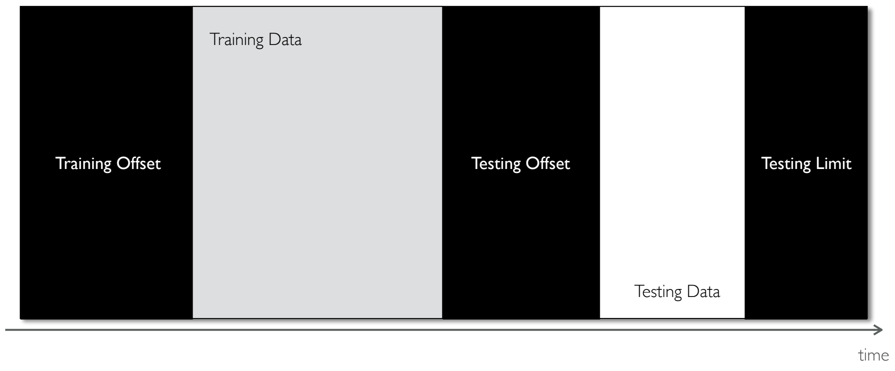


Figure 4.27.: Refined data partitioning for predictive modelling with gaps

As the causes for events of interest are not equally distributed over time, we have to consider that for the different data partitioning schemes and during the evaluation the results from the predictive models.

The grouping of similar activities as discussed in Section 4.3.4 presents another refinement opportunity. Variability in the behaviour of developers may not be strictly temporally bound. Developers may be performing activities under similar circumstances at different points in time. Thus, while the overall behaviour of a developer may vary, predictive models based on groups of similar activities may provide further refinements. A conceptual overview on an abstract representation of artifact activity map is shown in Figure 4.28. Similar activities for the green developer are clustered in two groups, outlined in blue and purple. Developer-centric predictive modelling for the green developer will consider all activities of a developer up to a given point in time (red vertical line) as training data for predicting the outcome of all activities beyond that point in time (green arrow). Instead, by grouping similar activities of a developer, only the activities in a given group are considered as training data. Future activities for each developer are then associated with the most similar group and evaluated against the corresponding model for that group in order to predict their outcomes (blue and purple arrows). Each model reflects the specific circumstances associated with causes for events of interest for the corresponding group.

While the overall behaviours of different developers may be dissimilar, there may be similarities between groups of activities across developers and across projects. This enables further transfer opportunities. Similar to the transfer opportunities for developer-centric predictive models, a conceptual overview of the extended transferring opportunities based on groups of similar activities is shown in Figure 4.29. Considering the same constellation of developers across the two projects *A* and *B*, we identify the fol-

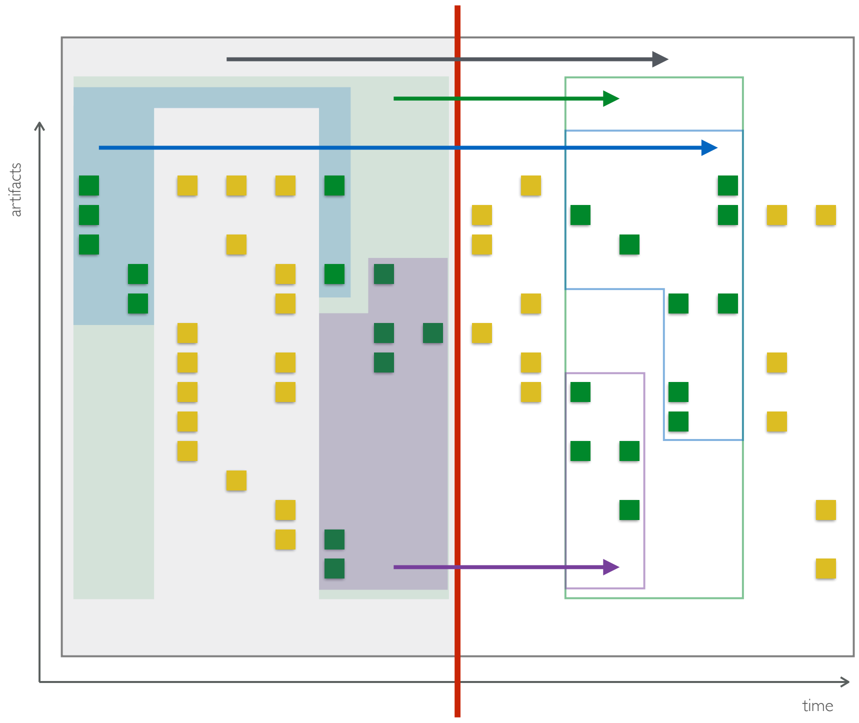


Figure 4.28.: Developer-centric predictive modelling with grouping of similar activities

lowing modelling and transferring opportunities based on groups of similar activities (for simplicity, we are assuming there are two groups of activities for each developer):

- using training data from group G_1 of developer *pat* in project A for predicting the outcomes of future activities within group G_1 of developer *pat* in project A (*same group, same developer, within-project*)
- using training data from group G_2 of developer *pat* in project A for predicting the outcomes of activities in group G_1 of developer *tom* in project A (*similar groups, different developer, within-project*)
- using training data from group G_1 of developer *sue* in project B for predicting the outcomes of activities group $G_2$of developer *tom* in project A (*similar groups, different developer, cross-project*)
- using training data from group G_1 of developer *joe* in project B for predicting the outcomes of activities in group G_2 of developer *joe* in project A (*similar groups, same developer, cross-project*)

Finally, to locate and better understand the prediction results in time and space, we can visualise them as an overlay on the artifact activity map. This way we can also

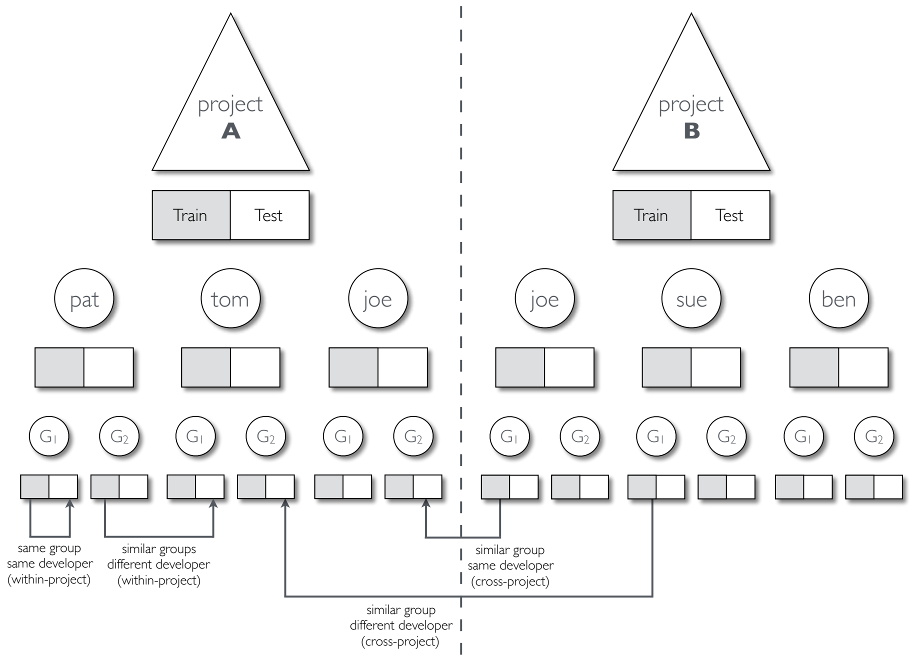

project A
project B
Train
Test
Train
Test
pat
tom
joe
joe
sue
ben
G1
G2
same group same developer (within-project)
similar groups different developer (within-project)
similar group same developer (cross-project)
similar group different developer (cross-project)

Figure 4.29.: Transferring opportunities for groups of similar activities

qualitatively compare the outcomes of different prediction models. Consider the example from the *log4j* project shown on Figure 4.30. Using the first half of the recorded activities for the whole project for training and the second half for evaluation, correctly and incorrectly predicted outcomes of activities are indicated by means of green and red diagonal lines over the corresponding activities, respectively. The length of the line is proportional to the confidence in the predicted outcome. We can observe that most prediction errors seem to occur in activities performed at around the same time, for example when multiple artifacts were changed at the same time and in particular when the first recorded activities on the artifacts were earlier in the recorded period. However, the opposite is not necessarily the case — there are also correctly predicted outcomes that fulfill the same criteria. The confidence in the predicted outcomes is generally high.

The prediction overlay can be refined further to also show the type of error (false positive or false negative). The breakdown of prediction errors is shown in Figure 4.31. False positives (type 1 errors), where an outcome of an activity was incorrectly predicted to be causing an event of interest, are shown in purple. False negatives (type 2 errors), for outcomes of activities incorrectly predicted not to be causing events of interest are shown in orange. The correctly predicted outcomes are not shown in this

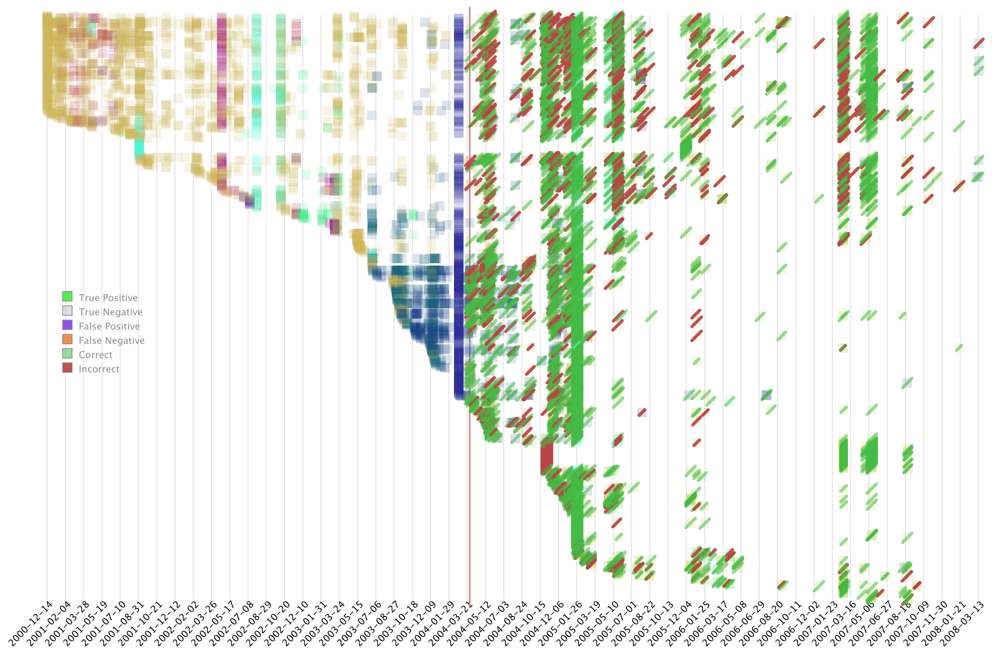


Figure 4.30.: Developer activities on source code files for log4j with prediction overlay

case. We can observe that, with a few exceptions, the errors for activities on artifacts for which there are no recorded activities in the training data are mostly false negatives.

4.4. Related Work

When it comes to defining developer behaviour, there are vastly different perceptions in the literature. Some are primarily concerned with program comprehension and code navigation [149, 192], where the authors study the facilities provided by IDEs and the extent to which developers make use of them in order to accomplish certain tasks. Others are more focused on the tasks and activities themselves [154] where the authors infer developer activities by recording observations from IDEs aiming to determine whether a developer is facing a problem, locating the cause of a problem, searching for a solution, or applying a solution. Still others study the behaviour based on the contributions of the developers, focusing on topics developers are working on. Linstead et al. [111] used statistical author-topic modelling to determine developer competences and demonstrated how that topic models can provide an effective basis for developer similarity analysis. Fritz et al. [51] found that the frequency and recency of interaction of a developer with parts of the code does indicate which parts of the code a developer

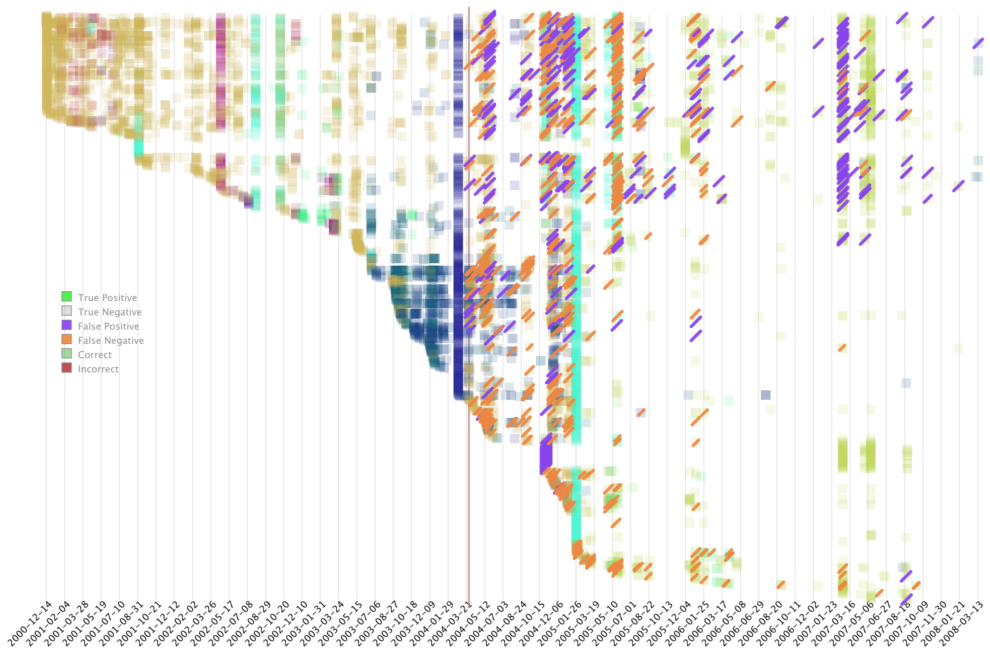


Figure 4.31.: Developer activities on source code files for log4j with error overlay

knows well based on interviews and mining. However, other factors such as authorship, the role of the code in the system, and the task of the developer play a role as well. Schuler and Zimmermann [161] discussed the concept of usage expertise based on the *Application Programming Interface* (API) calls associated with individual developers.

Girba et al. [57] discuss a comprehensive approach for identifying behavioural and collaboration patterns based on ownership maps, showcasing ten patterns. Vcavrak and Cmokvic [201] collected data from fourteen distributed student projects and identified a set of collaboration patterns and discussed their causes and implications. Dos Santos et al. [40] conducted an exploratory study targeting the identification and classification of characteristics of collaboration based on social networks properties such as centrality and density. Miranskyy et al. [123] proposed a temporal collaboration network model based on the history of collaboration among developers, testers, and other issue originators to estimate the defect exposure for the next month. Wiese et al. [186] present a systematic overview of the use of social metrics in prediction models in software engineering by conducting a mapping study. We discuss collaboration with regard to the notions related to this thesis. Some of the collaboration-related characteristics from the literature have been reused and refined. Future extensions of the approach may integrate further collaboration characteristics from the literature.

Different classifications of developer roles have been proposed previously in the literature. Von Krogh et al. [181] investigated the developer-initiation process in the Freenet project, based on interviews, mining e-mail exchanges and VCS repositories. They identified three roles, including *joiners* who joined the mailing list but do not have commit privileges, *newcomers* who have recently started contributing code, and *developers* who have been contributing core code to the project for longer periods of time. Lu and Ramaswamy [193] studied the interaction frequency of open source developers in two relatively small projects and derived two main roles for developers based on complete-linkage hierarchical clustering — *core developers* and *associate developers*. They also investigated which external attributes can be used to characterise the role of a developer and found that the percentage of modified lines and revisions can be good indicators for the role of a developer. For larger projects involving a large number of developers, they suggested further refinement of developer roles. In their topological analysis, Xu et al. [190, 191] identify three main groups of developers: *core developers* involved with significant contributions over long periods of time, *central developers* contributing regularly, and *peripheral developers* contributing fixes and new features irregularly. They note that a developer may belong to different groups in different projects, thus any classification can be considered project-specific. Nevertheless, the prevalence of different roles for a particular developer across all projects can still be considered as a characteristic of the developer. Honsel et al. [81] employed *Hidden Markov Model*s (HMMs) to model the dynamic activities and workload of developers according to the level of involvement and the role of different developers in a project by considering four characteristics. They considered only static implicit role assignment.

Given the high importance of determining which changes can be considered bug-introducing for any software project, there has been a large body of research dealing precisely with this problem within the field of defect prediction [4, 46, 127, 128] in software engineering and in particular focusing on the classification of changes as clean or buggy [95, 165]. Lumpe et al. [112] evaluated activity-centric measures in the context of *inspection optimization* [3] focusing on reducing the size of code to be inspected in order to find the most defects. However, they used the term activity in a different context — for detecting and measuring change of a class, where activity for a class is inferred by means of its volumetric and structural properties. Matsumoto et al. [118] investigated the effects of developer-related characteristics on software reliability. They found that such characteristics are a good predictor of software quality and indicated a need for considering further human factors for improving software reliability. The impact of code ownership and developer experience with certain artifacts have been investigated as well. Rahman and Devanbu [144] considered the impact of code ownership and developer experience on software quality. Their findings indicated that quality control efforts could benefit from targeting changes made by developers with limited prior experience on a given artifact. Bird et al. [15] found that high levels of ownership are associated with fewer defects. Posnett et al. [141] sought to unify the related notions of developer focus and artifact ownership and found that more focused developers tend to introduce fewer defects than developers which are not focused, while at the same time artifacts that receive narrowly focused activity are more likely to contain defects than other files. Bernstein et al. [12] proposed the use of non-linear models for defect prediction and argued that temporal aspects of data can be useful in improving prediction accuracy. While we make use of some of the characteristics discussed in the literature, we also define additional characteristics and discuss a refined conceptual view on characterising developer behaviour. We may integrate additional characteristics found in the literature in the future. The various related approaches are concerned with different levels of granularity, which are usually considered in isolation. Our approach seeks to provide a unified framework for integrating artifacts at different levels of granularity.

Two recent contributions [84, 163] described approaches of building separate developer-specific defect prediction models for the purposes of change classification. Shihab et al. [163] focused on risky rather than buggy changes, that is whether additional attention is needed for review and/or testing, regardless of whether or not the changes introduce bugs. They relied on industrial data which was manually annotated at commit time. The annotation was based on the intuition of the corresponding developers, reflecting the estimated uncertainty of changes with regard to delays and/or poor user satisfaction. They identified different characteristics as determining factors for individual developers and an overall improvement from using developer-specific predictive models. Jiang et al. [84] proposed *personalised defect prediction* based on predictive models for individual developers in order to account for different coding styles, contribution frequencies, and individual experiences. In their evaluation, they found significant improvements when using personalised defect prediction. They noted that data for at least 80 changes per developer are required for training personalised predictive models. These developer-specific defect prediction models seek to account at least partially for the different developer behaviours and highlight different factors contributing to increased defect likelihood for individual developers. This chapter presents further refinements of the approaches presented in these contributions, in particular transferring opportunities are also considered.

Applications of transfer learning techniques [140] for reusing of predictive models across projects have recently emerged as a new research direction in software mining. Approaches for transferring predictive models across whole projects have been investigated and evaluated in the literature in the related areas of cross-project defect prediction [74, 198] and cross-company defect prediction [176]. We pursue a refined approach for transferring predictive models for individual developer within the same project as well as across projects.

Agent-Based Modelling (ABM) [130] of software development processes has been gaining increasing attention. By definition, it requires a description of developer behaviour on some level. Wickenberg and Davidsson [185] were among the first to investigate the applicability of ABM for simulating software development processes. They provided a set of general guidelines concerning when to use ABM as well as concrete examples where ABM seems particularly promising. Dalle et al. [34] studied the mechanisms of allocating code-writing efforts within open source projects. They first described them analytically in a discrete choice framework, and then simulated them ABM experiments In a more refined approach, Smith et al. [167] also considered the complexity of software modules, the fitness of the software with regard to the requirements and the motivation of developers. In separate work, they also considered the modelling role of users in their simulation models [166]. Stopford and Counsell [170] outlined an even more sophisticated approach for simulating the structural evolution of software with stateful agents, evolution policies, and change operators. Zhang et al. [196] performed a systematic review of software process simulation modelling. Honsel et al. [80, 82] discussed refined ABM for simulating software processes based on the behavior of different types of developers. The characteristics and insights discussed in this chapter can be utilised to refine ABM for software development process simulation.

4.5. Summary

In this chapter, we discussed a conceptual overview of the approach to characterising developer behaviour based on the notions of situational and dispositional factors, as well as collaborative factors. A selection of characteristics across different dimensions was presented as basis for measurement. The measurements for the characteristics can result in vast amounts of data. We discussed different approaches for navigating and understanding the data, based on visualisation and data mining techniques. While the conceptual approach to characterising developer behaviour is novel, many of the individual characteristics have already been discussed in existing research and some have also found their way into practice. The selection of characteristics discussed in this chapter is used as basis for illustration and evaluation, however, the nature of the overall approach permits the use of essentially any other set of characteristics related to the artifacts and activities the developers are involved in. Generalising even further beyond the scope of this thesis, the approach can be used to systematically describe the behaviour of any type of entities performing activities on certain artifacts.

Sequential pattern analysis can be used to discover frequent sequences of activities associated with particular outcomes such as introducing a defect, or successfully implementing a new feature. Complementary to the sequential pattern analysis, clustering approaches can be used to segment activities into groups with similar circumstances and/or similar outcomes. Clustering can be applied to produce higher level characteristics which can then be used for sequential pattern analysis. While some clusters may not appear to be very interesting at first sight, combining properties that stand out with the outcomes of sequential patterns and classification results can produce added value.

5. Model-based Software Mining

For the realization of the approach for the identification of potential causes for events of interest described in Chapter 3 and the characterisation of developer behaviour described in Chapter 4, we need an appropriate infrastructure to manage all the necessary data extraction and processing steps along the way from raw artifacts to derived knowledge. In this chapter, we first contemplate the conceptual overview of the process and identify challenges arising along the way. These are then addressed in a generic model-based approach to software mining. A concrete instantiation of the model-based approach seeks to address the needs and challenges of software mining in the specific context of this thesis. This chapter is based on an extended and revised version of [116].

5.1. Mining Challenges

Software repositories store and organize software-related artifacts throughout the lifespan of a software project, maintaining successive revisions of the artifacts over time. Beyond source code files and related assets, software-related artifacts also include issue reports, development and user mailing list messages, and even user discussion forums.

Software mining is the process of extracting useful information from software repositories and software-related artifacts. This information typically comprises:

1. basic primary facts about the artifacts, including artifact histories and changes, as well as measurable attributes of artifacts and changes, and

2. derived knowledge resulting from assessment of the basic facts, such as defect prediction for artifacts and changes, pattern-detection, etc.

The extraction of basic facts and derived knowledge are also referred to as *data extraction*, and *synthesis* or *application*, respectively. The mined information is used for a number of assessment tasks and other purposes, such as guiding software development [200], detecting faults [139], predicting defect-prone artifacts [128] and changes [95], as well as assessing issue report reopening [199], costs [11] and risks [163] of development, etc.

The value and usefulness of mined information often depends on the context it is intended to be used in. Context in general describes the unique circumstances in which software mining is applied. The large variability in the different contextual dimensions has so far also resulted in a wide variety of context-specific methods and tools. These are often on a rather low level of abstraction and difficult to adapt to a different set of circumstances. This is often due to a tight coupling between data extraction and application.

This is one of the reasons researchers often resort to publicly available benchmark data sets, such as the PROMISE repository [121]. Such data sets provide a common ground of basic primary facts for the purposes of developing, evaluating, and comparing methods for knowledge derivation. While this allows for a relatively low-effort entry into the MSR field and for easy comparison and replication of method evaluation, there are inherent limitations in relying on such ready-made data sets. First, traceability to the original artifacts from which the data sets have been mined is difficult or impossible. This limits further investigation and validation of obtained results. Second, the available data is severely limited, and further potentially useful information, or information at finer levels of granularity, cannot be easily added to the existing data sets, due to the first limitation. Third, it is difficult to transfer the methods to other projects, without building the necessary infrastructure to produce the necessary basic facts in at least the same scope as the ready-made data sets.

Building the necessary software mining infrastructure presents a number of challenges of its own, and as a consequence collecting the necessary data can be both time consuming and computationally intensive. In [120], the authors note that most of data mining is actually data pre-processing. That is, before the data can be used for learning or other applications, a lot of effort is dedicated to selecting and accessing the data to process, pre-processing, and transforming it into a suitable form for the application in question. Due to the cyclic nature of data mining, finding one pattern related to a given question usually prompts new questions, and each question refines the goals of the data mining. This in turn leads to another cycle of the data mining process. This cyclic nature requires the process to be easily refined and repeatable in an agile stepwise manner without the need to dig through technical details every time. Furthermore, the authors note the frequent “quirkiness" of real-world data, often requiring the application of different methods before in order to identify an adequate approach for finding patterns in the data. Hence, at different stages in the application of data mining, different assessment applications based on visualisation, clustering, prediction, or simulation may come in question.

The starting point for a software mining infrastructure is usually a set of *raw assets* [62], i.e. source code files, revisions of source code files stored in VCSs, reported issues stored in ITSs, mailing list messages stored in *Mailing List Archive*s (MLAs), etc. In their raw form, these assets are often only suited for (mostly manual) qualitative assessment. Thus, they usually need to be processed in some way in order to obtain a set of *basic facts* [18] about the assets, which describe in a structured manner different attributes of the assets that are relevant to assessment tasks in a given context, both qualitatively and quantitatively. An example for such processing is calculating software metrics or detecting duplicates in source code. In practice, this process is further complicated by the heterogeneity of the raw assets, i.e. the source code may include components in different programming languages, different components may be managed with different VCSs, issues related to these components may be reported across different ITSs, etc. While there may be tools and methods already available for extracting some basic facts from these heterogeneous raw assets, using different tools may further exacerbate the problem of heterogeneity. On the one hand, different tools may extract different sets of facts for the same type of raw asset, e.g. there may be different measurable attributes for different programming languages, leading to heterogeneity in the structure and content of the extracted facts. On the other hand, different tools may extract facts for different types of assets in different formats, and this often applies even for the same type of raw assets, leading to heterogeneity in the representation of the extracted facts.

Managing heterogeneity of raw assets and basic facts is one challenge that gains even more importance with the need to integrate related facts. Facts obtained from different assets by different means may contain implicit relationships that may need to be reconstructed and made explicit. For example, a revision in the VCS may refer to an issue from the ITS that has been addressed in the revision or an issue may refer to the revision in which it has been addressed. Furthermore, the importance of different relations may vary across different assessment tasks and applications. On the other end of the knowledge derivation chain, similar to the tools and methods for extracting basic facts from raw assets, existing tools and methods for generating knowledge may also rely on heterogeneous *assessment assets*, both in terms of structure and in terms of representation, which presents similar challenges.

5.2. The Case for Model-based Mining

Bridging basic facts and the application tools and methods to generate actionable knowledge are *assessment tasks*. An assessment task starts out as a set of relevant concepts and relationships between them form the assessment domain that a practitioner is interested in. The high-level concepts may not necessarily reflect all the details available as basic facts, and they may not be directly related to the assessment assets expected by the tools and methods for generating knowledge. In practice, these high-level concepts and assumptions about them are often only implicitly or informally defined, and an actual realisation of the assessment task reflects them either only partially or not at all. A *model-based assessment task* on the other hand, seeks to formalise the relevant concepts and make them explicit throughout the realisation of the assessment task, in order to make the realisation understandable, traceable, extensible, and, most of all, allow practitioners to focus on the relevant concepts rather than the technical details of the underlying representations of both basic facts and assessment assets.

Model-based approaches such as MDE and supporting technologies enable practitioners to focus on higher-level domain-specific concepts, and the relationships among them, rather than on implementation details. The relevant concepts and relationships are typically described by means of *domain-specific meta-models*. This core principle is also one of the main strengths of the model-based approaches — the meta-model is the single point of truth with regard to the structure of domain-specific concepts and their relationships. Different concrete representations can then be mapped to the meta-model, enabling the interchange of model instances from one concrete representation to another, while still relying on the underlying common information model. Technologies supporting a given modelling framework enable the validation, storage, visualization, and transformation of all model instances conforming to meta-models described by using that modelling framework.

A model-based approach to software mining aims to provide a framework that relies on homogeneous high-level domain-specific models of facts extracted from raw assets. These facts models can then be combined and enriched by means of stepwise transformations into domain-specific assessment models related to particular assessment tasks. The domain-specific assessment models serve as a bridge between the data extraction and the assessment applications. The model-based approach seeks to address pragmatic challenges of extracting and integrating necessary data, while decoupling the data extraction from the application steps. This enables mining methods to be transferred to different contexts, including in-house assessment scenarios, and thus it lowers the barrier to entry for researchers and practitioners. Instead of integrating all available facts into one superstructure, practitioners can mix and match different domain-specific model instances at a high level of abstraction according to the needs of the specific assessment task at hand. The approach can be considered conceptually similar to Lego building blocks, which can be assembled according to a specific purpose.

5.3. Mining Process and Framework

To address the challenges related to the realisation of software mining infrastructures, we outline a framework that relies on defining domain-specific meta-models representing the structure and relationships of concepts related to the assessment tasks at hand. The approach serves as a glue between different existing third-party and custom-made mining solutions, interconnecting the various tools and assets, related to both data extraction and knowledge derivation, at a high level of abstraction, without unnecessarily tight coupling.

Software mining processes described in the literature commonly involve the following fundamental steps: data extraction, data modelling, synthesis, and analysis [72]. In the context of the present model-based approach, we refine these steps as follows (also shown as a visual overview in Figure 5.1):

Domain Modelling. When confronted with an assessment task, following a model-based approach, a practitioner would naturally first consider the concepts that are subject to assessment and how they are related among each other. This would form the basis for the conceptual domain model for the assessment task, serving as the foundation for the core of the proposed approach — the assessment meta-model resulting from the formalisation of the conceptual domain model. Depending on the intended level of abstraction, this assessment meta-model may be directly or indirectly related to the available facts concepts. The facts concepts themselves need to be formalised as well in the form of facts meta-models. While the foundations for both the assessment and the facts meta-models need to be defined in the beginning, the domain modelling step may ultimately span the entire process.

Facts Extraction. The facts extraction step handles processing of heterogeneous raw assets, such as VCS repositories and logs, ITS databases, source code files, and MLAs, in order to obtain basic facts and meta-data about these assets in a structured machine- (and human-) readable format. A number of existing approaches and tools already provide the necessary facilities for this step. The level of abstraction may vary, but the resulting assets are generally at a lower level of abstraction. In addition, since the resulting facts assets are usually extracted by different means, independently from one another, they are usually also not linked to each other, even if they refer to the same underlying raw assets and conceptual entities.

Facts Translation. Extracted facts assets are typically heterogeneous in both structure and representation format. In order to work with multiple assets using different representation formats, it becomes necessary to either translate all the assets to the lowest-common denominator format, or create a common access layer on top of the assets, which would require a common means for the description of the asset structures. Meta-modelling provides such means, where domain-specific meta-models describing the structure of the corresponding facts assets are created based on existing data structure descriptions or derived (automatically where applicable) from the facts assets. The concrete representation formats are then mapped to the corresponding meta-models, enabling the translation of the heterogeneous facts assets into homogeneous facts model instances conforming to the corresponding meta-models, resulting in a common high-level access layer.

Facts Transformation. Even with a common format or access layer, the relevant concepts in the different facts model instances need to be linked to each other and possibly mapped to corresponding higher-level concepts with respect to the particular assessment task at hand. This could be done during the translation into a common format or within the common access layer, but introducing such cou-

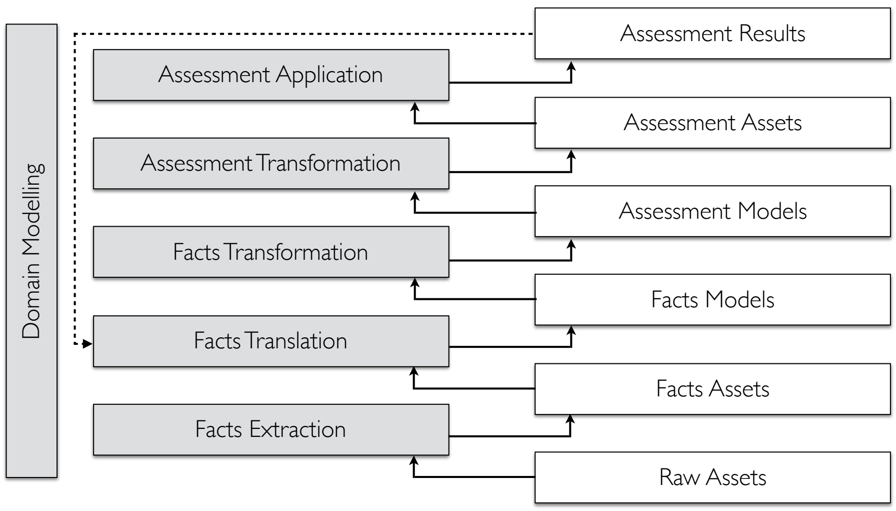


Figure 5.1.: Mining process overview

pling may restrict the flexibility of the approach as the assessment tasks evolve or different assessment tasks need to be considered. In our approach, we link related concepts from the different facts models instances during the transformation from facts model instances into assessment model instances according to the assessment task in question. This enables assessment-specific linking of relevant data only, as well as stepwise enrichment of the assessment model instances as new facts become available or necessary. Facts transformations can also be used to derive new facts from existing facts while still working at the model level.

Assessment Transformation. After the assessment-specific model instances have been derived from the facts model instances through stepwise transformation, queries and transformations over the assessment model instances are used to answer assessment related questions directly, or to produce application-specific assessment assets.

Assessment Application. The assessment assets are fed into corresponding assessment-specific applications, such as clustering, prediction, simulation, and visualization applications, which produce assessment results assets containing derived knowledge. These results assets can then be used as new facts assets, translated into new facts models, and integrated back into the assessment model, for example using cluster assignments for defect prediction, or for visualisation.

In the following sections, we contemplate a concrete instantiation showcasing the application of the model-based mining approach and addressing the challenges related

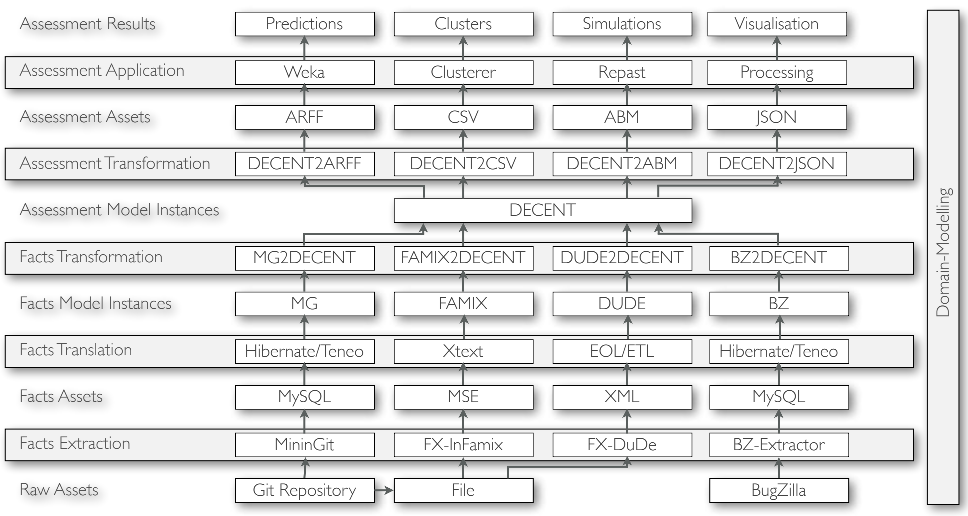


Figure 5.2.: Model-based framework instantiation overview

to the realisation of the approaches for finding events of interest (described in Chapter 3) and characterising developer behaviour (described in Chapter 4).

5.4. Instantiation

A high-level overview of the concrete instantiation of the model-based software mining framework is shown in Figure 5.2. The overview depicts the main steps decomposed into individual activities used for processing and transforming the different kinds of raw assets on one end and providing input for the assessment applications on the other end. These are interleaved with the intermediate assets and models resulting from each step for each raw asset and each application. It shall serve as a road map for the following sections.

5.4.1. Domain Modelling

As noted in Section 5.3, we first contemplate the domain under study and identify the relevant concepts that we are interested and the relationships among them. We first contemplate the concepts related to Chapter 3. These include *States*, differentiated into *Project States*, *File States*, and *Logical States* for the different levels of granularity, as well as *Factors* aggregating *Values* of *Attributes*, such as weights, related to a particular factor. The concepts are visually summarised in Figure 5.3.

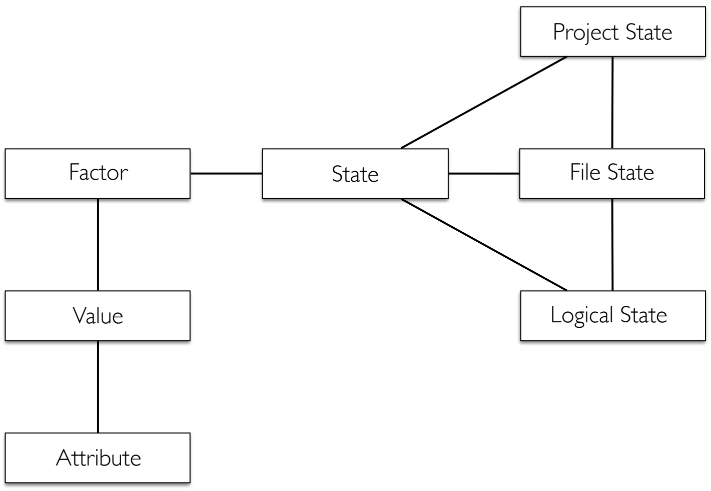


Figure 5.3.: Cause-fix-analysis — conceptual overview

Once we have identified the relevant concepts and their relationships, we proceed to formalize them by implementing them in a modelling framework, resulting in a corresponding meta-model. This process involves translating the concepts and relationships and adding necessary additional information, such as the nature of the relationships — e.g. direction, containment, multiplicities, names, etc., and characteristics of the concepts — e.g. names, descriptions, values of attributes, etc. Additional concepts and relationships may need to be introduced in order to describe further characteristics necessary for the realisation of the meta-model or make working with instances of the meta-model more convenient.

For the modelling tasks, we rely on UML [132] and the EMF[18] [19] which provide the necessary facilities for meta-modelling. The EMF provides an implementation of OMG's EMOF enabling pragmatic realisation of modelling and meta-modelling tasks. The meta-models are represented by using a subset of the UML *Class Diagram*, where relevant concepts are defined as *Classes* and relationships between them are defined as *Associations*.

The resulting meta-model for the cause-fix-analysis (or *CFA* for short) is shown as a class diagram in Figure 5.4. It reflects the conceptual overview with a meta-class corresponding to each concept. The relationships between the different meta-classes refine the relationships between the concepts. Notable refinements include:

- Abstract *State* meta-class that has possibly empty sets of *causing* and *fixing* states.

[18]See `http://www.eclipse.org/modeling/emf/`

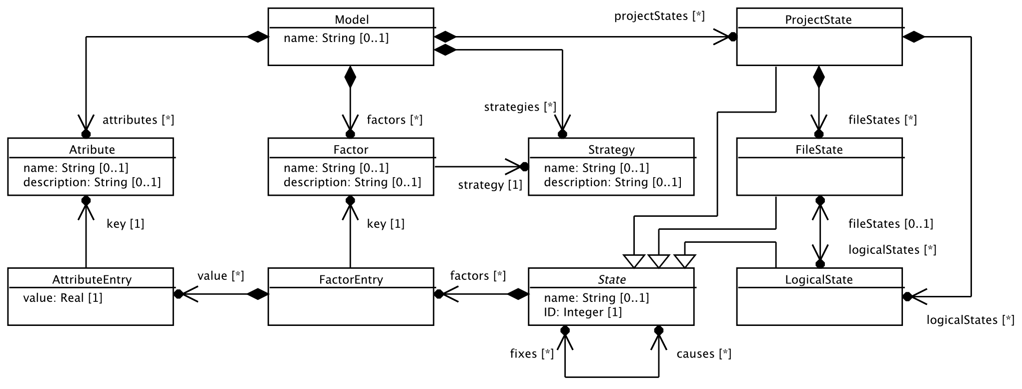


Figure 5.4.: Cause-fix-analysis — meta-model

- *ProjectState*, *FileState*, and *LogicalState* refine the abstract *State* meta-class, where a *ProjectState* may contain any number of *FileStates* and *LogicalStates*, and any number of *LogicalStates* may be associated with a *FileState*. Intuitively, one would expect *LogicalStates* to be contained within *FileStates*, however in practice this information may not be available depending on the language and tooling being used, thus the containment relationships have been adjusted to accommodate such cases.
- A *State* has a possibly empty set of *FactorEntries* which describe a relationship between a *Factor* and a possibly empty set of *AttributeEntries* comprising the *Values* assigned to the different *Attributes* for a given *Factor* in a given *State*.
- *Factors* are associated with a *Strategy* indicating how the *Values* assigned to the different *Attributes* for a given *Factor* in a given *ProjectState* shall be distributed across the corresponding *FileStates* and *LogicalStates*.
- Both *Factors*, *Attributes*, and *Strategies* have a *name* and a *description* informally describing their semantics.

Next, we contemplate the concepts related to Chapter 4, which are related to the assessment task of interest. We are concerned with a developer-centric assessment task, where conceptually the behaviour of a *Developer* is defined as a sequence of *Activities* performed on different software-related *Artifacts*. *Developers* and *Artifacts* are described by a sets of *Attributes*. Since both *Artifacts* and *Developers* evolve after each *Activity* in that the *Values* for the respective *Attributes* change after each *Activity*, both *Artifacts* and *Developers* have a sequence of *States* which is extended with each *Activity*. *Artifact States* are then described by a set of quantitative and qualitative *Values* of the *Attributes* characterising the corresponding *Artifact* at a certain point in time.

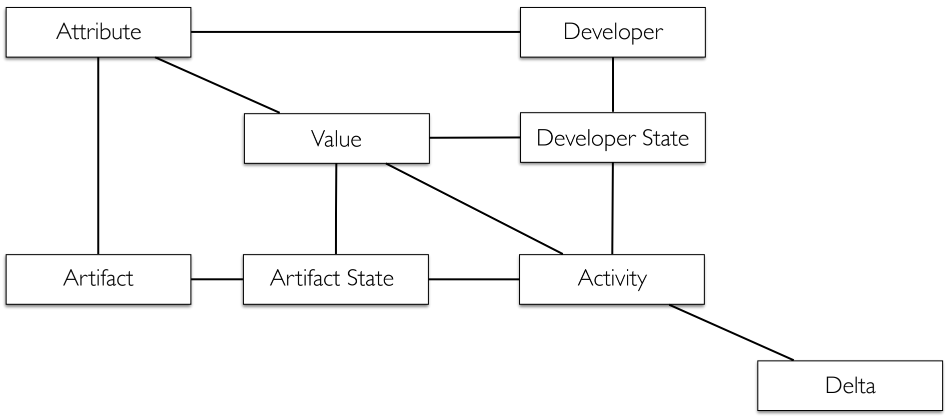


Figure 5.5.: Developer-centric software assessment — conceptual overview

Similarly, *Developer States* are described by a set of *Values* that characterise the corresponding *Developer* at a certain point in time, indicative of their experience at that point. Consequently, an *Activity* is performed in a given context defined by the *State* of the *Artifact* on which it is performed and the *State* of the *Developer* performing the *Activity*, resulting in a new *State* for the *Artifact* and potentially also for the *Developer*. *Activities* are described by a set of *Deltas*, which characterise the quantitative changes between the *State* of an *Artifact* in which the *Activity* was performed and the *State* resulting after the *Activity* was performed. *Activities* are also described by a set of *Activity Values* of *Attributes* which are related to the transition between the *States*. An visual overview of these concepts is shown in Figure 5.5.

Similar to the concepts related to the description of cause-fix relationships, once we have identified the relevant concepts and relationships, for the characterisation of developer behaviour, we proceed to formalize them by implementing them in a modelling framework, resulting in a corresponding meta-model.

The core-part of the resulting meta-model for the developer-centric (or *DECENT* for short) assessment task of interest is shown as a class diagram in Figure 5.6. It reflects the conceptual overview very closely with a meta-class corresponding to each concept. The relationships between the different meta-classes refine the relationships between the concepts. The gray coloured associations in Figure 5.6 are not strictly necessary and may be inferred indirectly. They are included as convenience shortcuts to make the use of the meta-model easier. Notable refinements include:

- An *Activity* is associated with 0 to 1 (source) *ArtifactStates* on which it is performed — the initial *ArtifactState* is also the result of an *Activity*, which is not performed on any pre-existing state, hence the 0 to 1 multiplicity.

- Conversely, an *ArtifactState* is associated with 0 or more *Activities* which are performed on that state. This reflects the fact that different development branches may result in multiple *Activities* being performed on the same *ArtifactState*. Also, multiple *Activities* on an *ArtifactStates* may be performed as a result of copying, where the first *ArtifactState* of the newly created copied *Artifact* is associated with the *ArtifactState* of the *Artifact* from which it was copied by means of the copying *Activity*, while there may still be further *Activities* performed on the *ArtifactState* from which the copy originated.
- An *Activity* is associated with 1 target *ArtifactState* resulting from the *Activity* being performed
- Conversely, an *ArtifactState* is associated with 1 or more *Activities* which it is the result from, where an *ArtifactState* may be the result of merging different development branches of an artifact, hence there may be multiple *Activities* resulting in that *ArtifactState*.
- A *Delta* is associated with an *Attribute*, which also serves as an indication of the corresponding *Values* in the source and target *ArtifactStates* related to the *Activity* containing the *Delta*.
- A *Delta* is associated with 1 target *Value* and 0 to 1 source *Value* as a shortcut representing the corresponding *Values* for the *Attribute* of the *Delta* in the source and target *ArtifactStates* of the *Activity* containing the *Delta*, respectively. The difference between the source and target *Values* is represented by the *Delta*, where for an *Activity* resulting in the initial *ArtifactState* of an *Artifact*, the source *Value* is not defined, in which case the *Delta* has the same content as the target *Value*.
- The generic notion of *Artifact* captures different levels of granularity, such as projects, files, classes, methods, etc., where an artifact may contain arbitrary number of other *Artifacts* as children.
- An *ArtifactState* is associated with 0 or more previous and next *ArtifactStates* as a shortcut for *ArtifactStates* that may be reached via *Activities* performed on or resulting in an *ArtifactState*.
- An *ArtifactState* is associated with 0 to 1 parent *ArtifactStates* as a shortcut for the *ArtifactState* of the parent *Artifact* of the corresponding *Artifact* at the same point in time.
- Conversely, an *ArtifactState* is associated with 0 or more children *ArtifactStates* as a shortcut for all the *ArtifactStates* of the children *Artifacts* of the corresponding *Artifact* at the same point in time.
- A *DeveloperState* is associated with 0 to 1 previous and next *DeveloperStates* as a shortcut for the *DeveloperStates* that temporally precede or follow the *DeveloperState*.

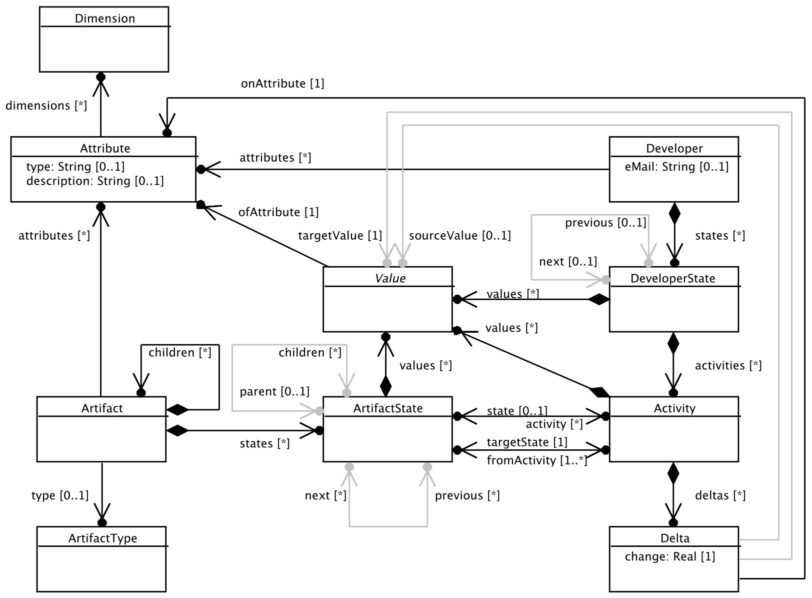


Figure 5.6.: Developer-centric software assessment — meta-model (core part)

In addition to the meta-classes directly related to the concepts from conceptual overview in Figure 5.5 described in the core part in Figure 5.6, some further meta-classes are needed to refine some of the concepts. The *Dimension* meta-class associated with the *Attribute* meta-class enables the classification of *Attributes* according to various properties, such as their origin (e.g. static analysis, duplicate detection, etc.) and their nature (e.g. temporal, spatial, experience-related, etc.). The *ArtifactType* meta-class associated with the *Artifact* meta-class enables the classification of *Artifacts* according to their type and level of granularity (e.g. project, component, file, package, class, method, etc.).

The *Value* meta-class in the *DECENT* meta-model is defined as abstract in order to accommodate different kinds of values. The refinement of the *Value* meta-class is shown in FIgure 5.7, including, *string*, *integer*, and *real* values, as well as lists of these.

Finally, a top-level container meta-class *Model* is included to contain the *Attributes*, *Dimensions*, *ArtifactTypes*, *Artifacts*, and *Developers* as shown in Figure 5.8. This top-level container enables grouping all elements related to a particular model instance together.

All meta-classes inherit directly or indirectly from the abstract *Element* meta-class as shown in the complete inheritance hierarchy in Figure 5.9, with the *DeveloperState* and *ArtifactState* meta-classes inheriting from the abstract *State* meta-class and all value-related meta-classes inheriting from the abstract *Value* meta-class. The *Element* meta-

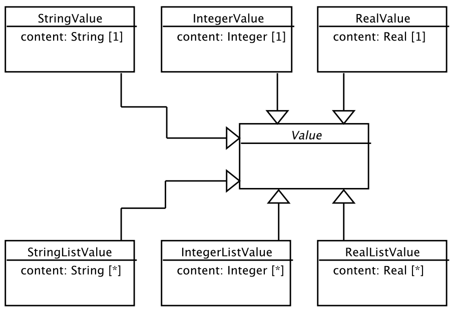


Figure 5.7.: Developer-centric software assessment — meta-model (values part)

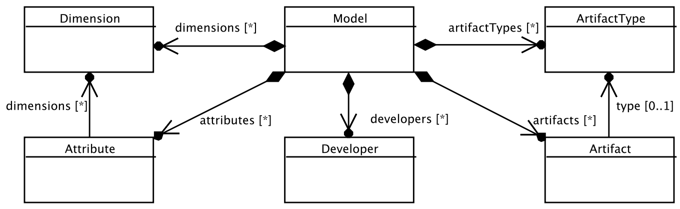


Figure 5.8.: Developer-centric software assessment — meta-model (top-level container)

class provides an optional *name* attribute and the *State* meta-class provides a mandatory *ID* attribute.

The purpose of the assessment meta-model in general is to abstract way from the peculiarities of facts extracted by different means from the available raw assets and enable practitioners to focus on the concepts related to the assessment domain of interest. This also applies to the *DECENT* meta-model which provides a rather generic framework able to accommodate information from different sources. After establishing and formalising the domain of the assessment task in a corresponding meta-model, we need to consider how we can create instances of the model. This also applies to the *CFA* model. For instances of both models we need to determine what information will be used, where this information can be obtained from, and how it needs to be transformed in order for it to fit into framework provided by the corresponding meta-model. We also need to consider how instances of both models can be integrated in order to incorporate information from the *CFA* model into the *DECENT* model. In the next sections we will look into how facts commonly extracted for assessment purposes are integrated into instances of the *DECENT* and *CFA* meta-models and define the necessary meta-models for the different kinds of extracted facts.

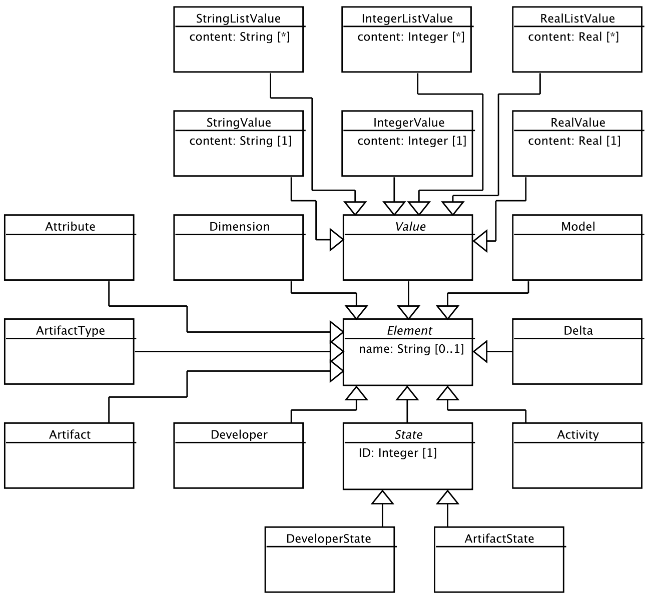


Figure 5.9.: Developer-centric software assessment — meta-model (inheritance part)

5.4.2. Facts Extraction

The facts extraction is concerned with processing the raw assets by means of facts extractors, resulting in facts assets. Concrete raw assets considered in our instantiation are a *Git VCS repository* containing revisions of *files* and other information related to the development history of the project and its associated artifacts, and a *BugZilla ITS repository* containing issue reports. For the extraction of basic facts, we employ readily available task-specific facts extractors, complemented by custom facts extractors where necessary.

5.4.2.1. VCS Repository Facts

For the VCS repository we rely on *CVSAnalY*[19] [153] and its fork *MininGit*[20] (also used by Lewis et al. [157]). *CVSAnalY/MininGit* processes VCS logs and stores extracted facts into a relational database (e.g. *MySQL*). The resulting facts are related to concepts common to the domain of VCS, such as *Repositories*, *Branches*, *Revisions*, *Files*, *Actions*, and *People*. Apart from the core functionality, a number of extensions

[19]See `https://github.com/MetricsGrimoire/CVSAnalY`,
[20]See `https://github.com/SoftwareIntrospectionLab/MininGit`.

for *CVSAnalY/MininGit* can be executed to obtain additional facts[21]. The available extensions are:

FileTypes providing information about the type of the file assets within the VCS repository, e.g. *code*, *documentation*, *image*, etc.

BugFixMessage providing information on whether a revision is considered a fix for an existing issue based on matching the commit message for the revision against a set of regular expressions[22], which can also be further customised by the user.

Content providing the complete content for file assets in each revision (for textual files only).

Patches providing information about the content of the fragments changed in a revision (for textual files only).

Hunks providing information about the location of the fragments changed in a revision, comprising the start and end lines for each fragment, before and after the change (for textual files only).

HunkBlame providing information about the revisions in which a fragment (hunk) was last changed (for textual files only).

LineBlame providing information about the revision in which a line or a fragment was introduced (or last changed) (for textual files only).

PatchesLOC providing information about the number of lines added and removed for each file changed in a revision (often used for calculating code-churn metrics).

CommitsLOC providing information about the number of lines added and removed in each revision as a whole (often used for calculating code-churn metrics).

Blame providing information about the number of lines changed in each file asset in each revision and the corresponding author (often used for calculating ownership metrics).

We extract all the available facts by executing all the available extensions (with the default regular expressions for the *BugFixMessage* extension), as we do not want to

[21] See `https://github.com/SoftwareIntrospectionLab/MininGit/blob/master/docs/miningit.mdown`.

[22] The default set of regular expressions includes:
```
"defect(s)?","patch(ing|es|ed)?","bug(s|fix(es)?)?",
"(re)?fix(es|ed|ing|age|\s?up(s)?)?", "debug(ged)?","\#\d+",
"back\s?out","revert(ing|ed)?"
```

limit the applicability of the extracted facts, even if we do not strictly need all of them for the concrete instantiation of the mining infrastructure.

In addition, due to limitations of *CVSAnalY / MininGit*, which does not extract the revision hierarchy, we also extract a Directed Acyclic Graph *(DAG)* of the revision hierarchy by means of a separate custom facts extractor (*DAG-GitExtractor*[23]), which produces a Comma Separated Values *(CSV)* representation of the revision branching hierarchy where each row contains a revision and its parent revision(s). Correct hierarchy is important for both getting the correct context in which an activity was performed, as well as getting the correct deltas describing the activity.

5.4.2.2. Static Code Analysis Facts

Static code analysis is an established field, where analysis tools are applied to extract statically computable facts about source code, including calculating software metrics, detecting duplicated code fragments and other anomalies (so called *code smells* [50]), analysing dependencies among artifacts, both on the logical and file level, etc. A number of commercial, open source, and research solutions exist for virtually every moderately popular programming language in use. The scope of the extracted facts varies among solutions and technologies, and also depends on the language features.

Within the scope of this thesis, we are primarily contemplating object-oriented software systems implemented in the languages C++ and Java, but the proposed approach is not limited to the object-oriented programming paradigm or the particular languages of interest and further technologies can be easily supported as long as there are tools available that can perform static code analysis and extract the necessary facts. For the static code analysis, we used the *InFamix*[24] facts extractor. *InFamix* processes C/C++ and Java code files and produces a *FAMIX 3.0* model instance for each revision in the *MSE*[25] format containing source code metrics, as well as structural and dependency information, which are associated with artifacts at the logical level of abstraction, represented by concepts such as *Classes*, *Methods*, *Inheritance*, *Attributes*, and *Invocations*.

A particular type of static analysis is duplicate detection. There are different approaches to duplicate detection taking into consideration different measures of similarity, and employing different approaches to detect duplicates. These also vary with respect to the required information and the supported languages and assets. We selected the *DuDe*[26] duplicate detector which employs a rather simple approach based on the concept of duplication chains. *DuDe* works on text-based assets and is language inde-

[23]The results can be effectively obtained also by using:
```
git log --topo-order --pretty=format:"%H %P" --parents
-M -C --cc --decorate=full --all > model.dagx
```
[24]See `https://www.intooitus.com/products/infamix`.
[25]See `http://www.moosetechnology.org/docs/mse`.
[26]See `http://www.inf.usi.ch/phd/wettel/dude.html`.

pendent and thus widely applicable. It produces duplication facts assets in *XML* format for each revision, containing facts represented as concepts common to the domain of code duplicates, such as (cloned) *Code Fragments* and *Clone Pairs*. An extension for *DuDe* was created to enable support for storing duplication facts in a relational database (*MySQL*) in addition to the *XML* format.

We applied the *InFamix* and the *DuDe* facts extractors to the files within each revision of the *Git VCS repository* by successively checking out the respective revision and running the facts extractors. This task was accomplished by a custom facts extractor automation framework (*FX*) which controls the overall process and manages the resulting facts assets. The extensible *FX*-framework supports arbitrary facts extractors operating at the revision level. Since the process can be very time consuming, and given that the facts that are extracted are largely independent at this stage, the *FX*-framework also supports distributed execution within a computing cluster by assigning sequences of revisions to be processed on individual compute nodes within the cluster. The *FX*-framework was deployed on a computing cluster of 40 nodes which reduced the processing times by a factor roughly equal to the number of nodes.

5.4.2.3. ITS Repository Facts

ITSs are collaboration tools used to create and manage issue reports by different stakeholders, such as users, managers, and even developers themselves. They serve as means for coordination and transparency, as well as enabling users to participate in the development of software systems. Issues may range from bug and error reports, to questions needing clarification, as well as requests for new features. ITS repositories store information related to the reported issues, often covering their entire life-cycle, from the initial report to their final resolution, including any modifications, assignments, and comments in between. As such, they provide further evidence describing the behaviour of developers, and often events and artifacts within an ITS repository may be related to events and artifacts within a VCS repository by means of semi-formal links between the underlying raw assets based on conventions, such as using references to issues within revision messages, or references to revisions within comments related to an issue.

There are a number of commercial and open source ITSs in use, sharing common and related concepts, yet differing in the details and the amount of information recorded. Even the same ITS may be configured in different ways for different contexts. As a consequence, extracting facts from ITSs may be a challenging task depending on the specific context, and facts extracted from one ITS may not necessarily be compatible with facts extracted from another ITS or even from a different configuration of the same ITS. As a result, facts extractors for ITSs often resort to the lowest common denominator approach by covering the essential and common concepts across different ITSs while leaving out details specific to a particular configuration or ITS. For our purposes, we used a custom facts extractor for the *BugZilla* ITS which enables us to extract more detailed information for specific contexts, at the expense of needing further adaptation for additional application settings. The custom *BZExtractor* is used to extract facts about issue reports from a *BugZilla repository* into a relational database (*MySQL*), representing concepts common to the ITS domain, such as *Products*, *Components*, *Issues*, *Comments*, as well as *Events* related to modifications of the meta-data of an issue.

5.4.3. Facts Translation

Due to the reliance on third-party facts extractors, the resulting heterogeneous facts assets in different formats and utilizing different storage paradigms are usually what software mining practitioners are confronted with once they get past the raw assets. As it is often the integration of these heterogeneous assets that practitioners are most interested in, the next challenge is figuring out how to achieve that. One way to approach this is to convert all facts assets in a common format, such as a relational database, but that approach is still too concerned with the concrete storage paradigm. Another approach is to lift the level of abstraction above the concrete storage paradigms and work at a homogeneous structural level with model instances representing the extracted facts rather than with their concrete storage representations. In this section we pursue the latter approach.

In this step, we translate the heterogeneous facts assets into homogeneous facts model instances conforming to a set of meta-models describing the structure of the facts. We first need define or derive (automatically where applicable) the meta-models for the facts, which would serve as the basis for the subsequent mapping and transformation descriptions. In Section 5.4.2 we already mentioned some of the essential concepts related to the domains of VCSs, static analysis, code duplication, and ITSs. By inspecting the structure of extracted facts more closely we identify all the relevant concepts and their relationships, and, based on these, define the corresponding domain meta-models. The meta-models may be based on existing specifications, if such are available, or they need to be reverse engineered if no suitable specification is available. After defining the meta-models, we need to also map the concrete representations of the facts assets to instances of the corresponding meta-models. The mapping may be based on available means, such as *Model-to-Text* (M2T) mappings for structured textual representations, *Object Relational Mapping* (ORM) frameworks for relational databases, or even custom-made mappings if no other means are available.

In the following sections, we discuss the concrete realisation of the facts translation steps for the various facts assets obtained during the facts extraction step.

5.4.3.1. VCS Repository Facts

While there are some descriptions for the data structure resulting from the application of *CVSAnalY/MininGit*, these are mostly informal for informative purposes and partially outdated or incomplete. The best and up-to-date description available is the database schema for the extracted facts. Hence, we decided to rely on it as a basis for the domain meta-model.

Based on the structure of the relational database produced by the application of *MininGit*, we derive the *MG* meta-model for representing VCS repositories shown in Figure 5.10. The *MG* meta-model is comprised of meta-classes for concepts closely related to VCSs (as noted in Section 5.4.2.1), with additional details for the relevant relationships between them, and their attributes. These include meta-classes for the core concepts *Files*, *Revisions*, *Branches*, *Patches*, etc., shown in the middle and the right part of Figure 5.10, as well as meta-classes for concepts related to the facts extracted by means of the optional extensions of *MininGit*, such as *Hunks*, *Patches*, and *Content*, among others, shown in the left part of Figure 5.10.

Once the domain meta-model for the extracted facts is defined, the next step is to define the mappings from the concrete representation to the meta-model. In the case of relation databases, this can be achieved by means of ORM frameworks, such as *Hibernate*[27] and *EclipseLink*[28]. Depending on the selected approach, the overhead of defining the mappings may vary. We selected a combination of *Hibernate* and *Teneo*[29] which provides automated and customizable model-relational mapping generation and good integration with the modelling framework with little overhead. As a result, the *MySQL* database storing the extracted facts is treated as a concrete representation of a model instance, accessible by a *Uniform Resource Identifier* (URI), just like any other concrete representation. If necessary, the relational database representation can be translated into a different concrete representation, such as an XMI representation which is commonly supported among model-based applications out of the box, or a more compact binary representation, in order to remove the dependency on the database. Details regarding the mappings can be found in Appendix A.2.

As noted in Section 5.4.2.1, a complementary facts extraction covering the revision hierarchy needed to be performed in order to obtain the correct context of changes in the case of multiple branches being used during the development. Since the extracted facts cover only a rather simple concept, the respective *DAG* meta-model shown in Figure 5.11 is also fairly inconspicuous, but also usable as a generic graph meta-model. It includes meta-classes for a *Graph* concept containing any number of *Nodes*, which may reference any number of parent and children *Nodes*, and *Edges*, where *Nodes* reference incoming and outgoing *Edges*, which in turn have back-references to the source and target *Nodes*.

Once the domain meta-model is defined, we proceed to define the mappings from the concrete representation to the meta-model. In the case of structured textual rep-

[27]See `http://hibernate.org`
[28]See `http://www.eclipse.org/eclipselink/`
[29]See `https://wiki.eclipse.org/Teneo/Hibernate/`

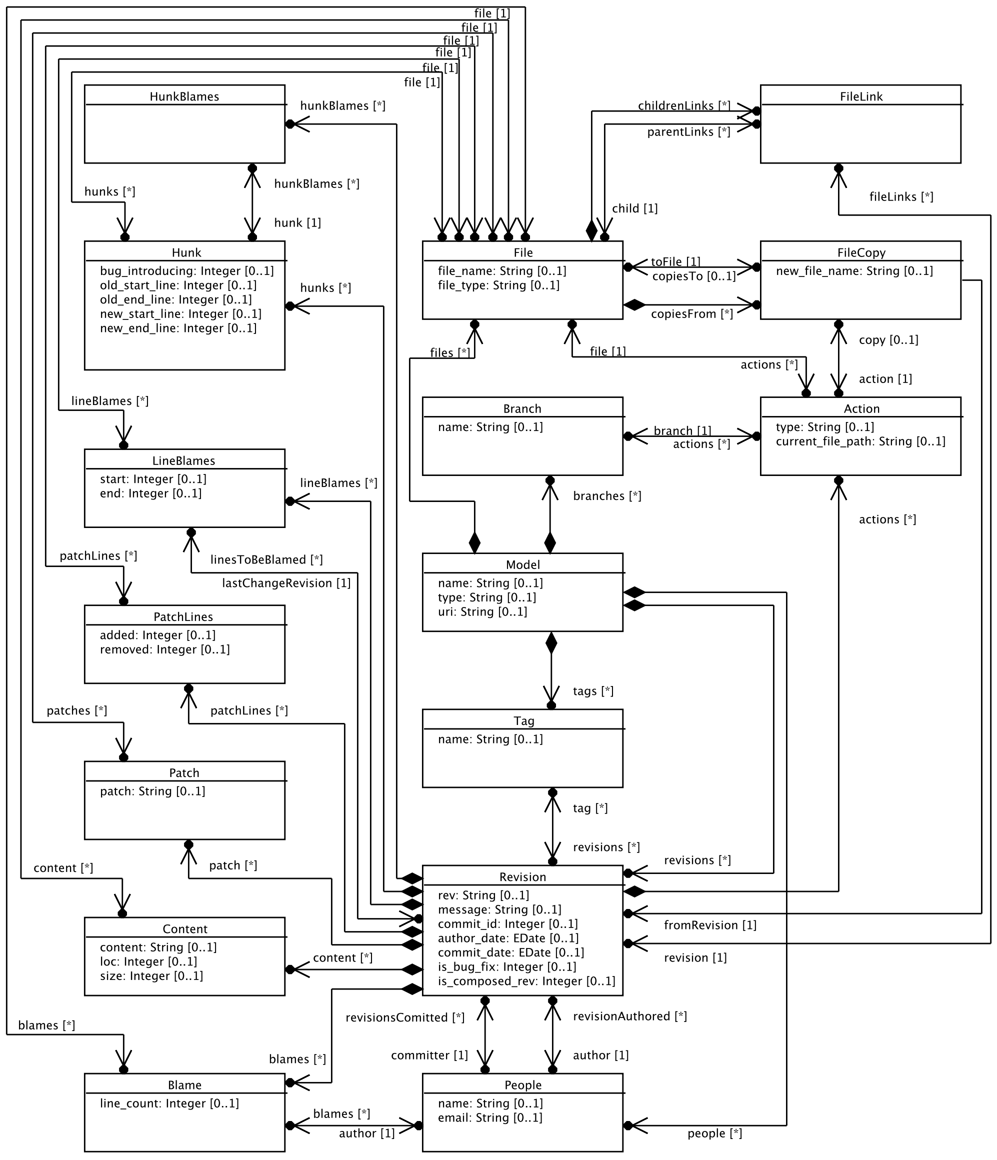


Figure 5.10.: *MG* meta-model for the structure of facts extracted with *MininGit*

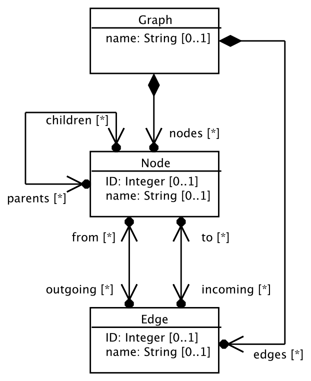


Figure 5.11.: *DAG* meta-model for the structure of facts extracted with *DAG-GitExtractor*

resentations, this can be achieved by means of a M2T transformation, which involves parsing the textual representation in to an AST, and reconstructing relevant relationships between the AST nodes based on static analysis. Available *Domain-Specific Language* (DSL) specification frameworks, such as *Xtext*[30] and *EMFText*[31], enable convenient high-level specification of concrete textual representations for a given domain meta-model. Such frameworks provide all the necessary facilities to serialise and deserialise model instances of the domain meta-model into the specified concrete representations. Based on previous experiences, we selected *Xtext* as the framework of choice. It relies on an annotated EBNF dialect for defining the concrete textual representation with annotations defining the mappings to the domain meta-model elements. The full annotated EBNF for the *DAG* meta-model can be found in Appendix A.2.

5.4.3.2. Static Code Analysis Facts

The *FAMIX 3.0* model instances in the *MSE* format resulting from the application of the *InFamix* facts extractor on each revision pose an interesting challenge. The text-based *MSE* format is used as the exchange format for Moose and related technologies. In many ways, it is similar to XMI, where it can be used for the serialisation de-serialisation of any *FM3* model instance.

Since our mining infrastructure is build around EMF, we need to derive the corresponding domain meta-model for *FAMIX 3.0* model instances and then define the necessary mappings from the concrete *MSE* representations to the instances of the *FAMIX* meta-model defined in EMF.

[30]See `https://eclipse.org/Xtext/`
[31]See `http://www.emftext.org`

While there is a grammar for the *MSE* format[32], the *MSE* format is intended for the interchange of all *FM3* compliant models and meta-models, hence the grammar is at a much higher level of abstraction (corresponding to *M3*), covering the serialisation of abstract elements and their attributes. De-serialisation of *MSE* assets according to the grammar would hence produce a very abstract representation of the underlying model or meta-model which needs to be interpreted further. This interpretation may be performed against a known meta-model enabling also structural and semantic validation during de-serialisation. In case there is no known meta-model, it may be inferred based on information collected from available assets. While there are descriptions of the *FAMIX 3.0* meta-model [44], we pursued the latter approach as an experimental feasibility study for integrating complex structured data in possibly proprietary formats.

Based on an initial pre-processing of available *MSE* assets, we inferred a possible meta-model structure, based on which we also inferred a specific concrete syntax tailored to the inferred meta-model and able to serialise and de-serialise instances of the inferred meta-model corresponding to the observations made on the available assets. This effectively comprises reverse engineering of the underlying meta-model and the necessary facilities to operate with assets from possibly proprietary third-party tools. The core of the inferred *FAMIX* meta-model is shown in Figure 5.12. It includes abstract meta-classes for concepts such as *Measurable Declared Types* and *Behaviour Entities* containing measurable attributes of these concepts. Refinements of these meta-classes include concrete meta-classes representing concepts for artifacts at the logical level of abstraction, such as *Classes*, *Methods*, and *Functions*, as well as other related concepts. The *File Anchor* meta-class indicates the location of the logical artifacts within artifacts at the file level of abstraction, thus providing a link to artifacts represented in the facts extracted from the VCS. The *File Anchor* meta-class is related to all *Anchored Elements* early in the overall inheritance hierarchy of the inferred *FAMIX* meta-model, as shown in Figure 5.13. Since the inferred meta-model is based on observations, it is inherently incomplete, and only covers concepts that have been already observed. In order to keep track of potentially interesting concepts or properties that have not been observed yet, a meta-class for the concept of *Water* (based in part on a similar notion from island grammars [125]) is introduced to store any unknown structural properties of known concepts or unknown concepts altogether. These can then be inspected and if considered useful incorporated into the meta-model iteratively.

The partial view on the reconstructed *FAMIX* meta-model shown in Figure 5.10 covers only the essential constructs that are of primary interest for the assessment task of interest. The reconstructed *FAMIX* meta-model includes a number of additional concepts that capture structural dependencies among logical constructs which are not shown here for brevity. The additional views on the reconstructed *FAMIX* meta-model are included in Appendix A.1.

[32]See `http://scg.unibe.ch/wiki/projects/archive/fame/msespecification`

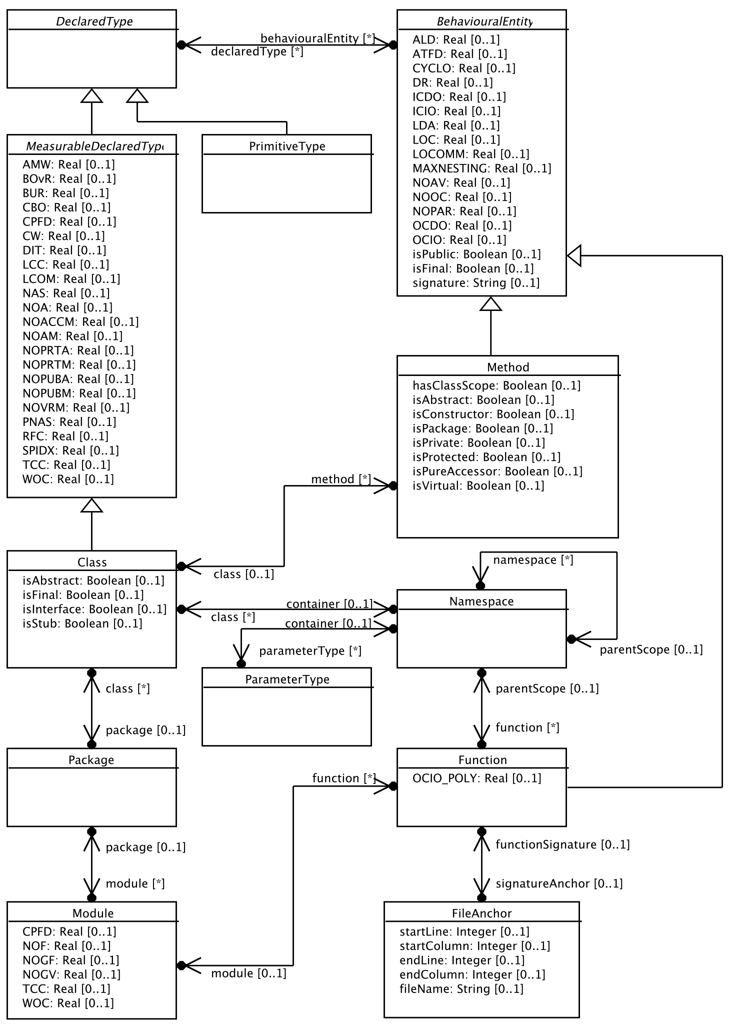


Figure 5.12.: *FAMIX* meta-model based on the structure of data extracted with *InFamix* (core-part)

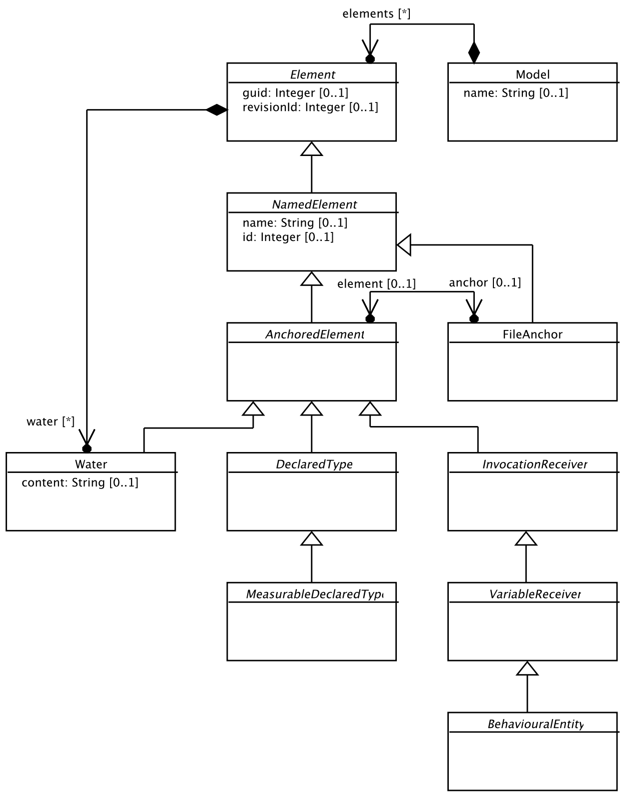


Figure 5.13.: *FAMIX* meta-model based on the structure of data extracted with *InFamix* (elements-part)

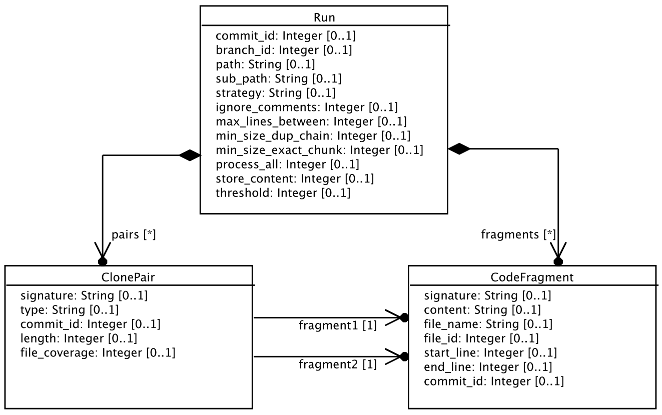

Run
commit_id: Integer [0..1]
branch_id: Integer [0..1]
path: String [0..1]
sub_path: String [0..1]
strategy: String [0..1]
ignore_comments: Integer [0..1]
max_lines_between: Integer [0..1]
min_size_dup_chain: Integer [0..1]
min_size_exact_chunk: Integer [0..1]
process_all: Integer [0..1]
store_content: Integer [0..1]
threshold: Integer [0..1]
pairs [*]
fragments [*]
ClonePair
signature: String [0..1]
type: String [0..1]
commit_id: Integer [0..1]
length: Integer [0..1]
file_coverage: Integer [0..1]
fragment1 [1]
fragment2 [1]
CodeFragment
signature: String [0..1]
content: String [0..1]
file_name: String [0..1]
file_id: Integer [0..1]
start_line: Integer [0..1]
end_line: Integer [0..1]
commit_id: Integer [0..1]

Figure 5.14.: *DUDE* meta-model based on the structure of data extracted with *DuDe*

Once the domain meta-model is defined, the next step is to define the mappings from the concrete representation to the meta-model. Similar to the handling of the revision hierarchy facts extracted with the *DAG-GitExtractor*, we employ a M2T approach based on *Xtext* for this purpose. The full annotated EBNF for the *FAMIX* meta-model can be found in Appendix A.2.

For the translation of the code duplication facts extracted with *DuDe*, we chose to rely on the relational database representation of the extracted facts produced by the custom extension, as noted in Section 5.4.2.2, which provides additional information and more flexibility. Based on the structure of the relational database, we derived the *DUDE* meta-model for representing concepts related to code duplication. A class diagram for the resulting *DUDE* meta-model is shown in Figure 5.14. The structure of the underlying facts is comprised of *Runs* containing *Clone Pairs* and (duplicated) *Code Fragments*, where a *Clone Pair* refers to exactly two *Code Fragments*. A *Run* typically represents the application of *DuDe* on a particular revision of a project, but it may also be restricted to a particular subset of the project as indicated by the *sub path* attribute. A *Run* also contains attributes for other parameters used in the application of *DuDe*, in case it was executed multiple times on the same revision with different parameters (e.g. using different detection strategies and/or thresholds). A *Code Fragment* contains attributes describing the location of the fragment and optionally also its content.

Once the domain meta-model for the extracted facts is defined, the next step is to define the mappings from the concrete representation to the model instances. As we chose to rely on the relational database representation of the extracted facts, the mapping approach is similar to the one employed for the mapping of the *MG* domain meta-model, based on a combination of *Hibernate* and *Teneo*. Details regarding the complete mappings can be found in Appendix A.2.

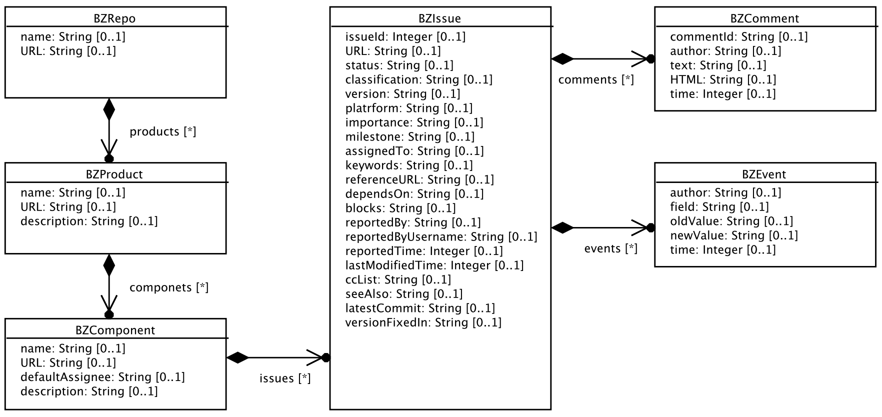


Figure 5.15.: *BZ* meta-model based on the structure of data extracted with *BZExtractor*

5.4.3.3. ITS Repository Facts

For the translation of the ITS repository facts extracted with the custom *BZExtractor* which stores the facts into a relational database, we follow the same steps as with other facts stored in relational databases. Based on the structure of the relational database, we derived the *BZ* meta-model for representing concepts related to ITSs. A class diagram for the *BZ* meta-model is shown in Figure 5.15. The structure of the underlying facts is comprised of *Repositories* containing *Products*, which in turn contain *Components*. The main model elements of interest are the *Issues* contained within the *Components*. Each *Issue* may contain any number of *Comments* and *Events* reflecting changes to the meta-information related to the *Issue*.

Once the domain meta-model for the extracted facts is defined, we proceed to define the mappings from the concrete representation to the meta-model. As the extracted facts are stored in a relational database representation, the mapping approach is similar to the one employed for the mapping of other facts stored in relational databases. Details regarding the complete mappings can be found in Appendix A.2.

With the steps discussed above, we can obtain homogeneous high-level model representations of the heterogeneous facts assets containing the facts extracted by third-party tools from the available raw assets. Once these are available, we can transform and integrate the facts models into instances of the assessment model.

5.4.4. Facts Transformation

Having obtained the high-level model representations for the extracted facts by means of the facts translation approaches described in Section 5.4.3, we can proceed and transform the relevant parts of the facts models into instances of the domain meta-model for the assessment task of interest. This is done in a stepwise manner, by means of M2M transformations. The overall transformation approach is based on EOL[33] and ETL[34] [98]. EOL is a domain-specific language for creating, querying, and modifying EMF models. It supports the access and modification of multiple models conforming to potentially different meta-models, by means of common programming constructs, as well as first-order logic OCL operations. EOL also provides a good integration of Java-based external tools which can be reused when working with models within EOL. ETL is a domain-specific language for hybrid, rule-based M2M transformations built on top of EOL. It provides common transformation capabilities, as well as the ability to transform many input to many output models, including modifying both source and target models in place. The transformations are defined by means of both declarative and imperative transformation specifications, allowing for sophisticated transformation logic, as well as abstraction and reuse. Traceability links between transformed elements can be recorded as well. While there are a number of other technologies for model transformation available, based on previous experiences, we selected the Epsilon family of languages as the most convenient solution. In the following we outline the essential steps in each transformation defined according to the following template:

SYMBOLIC NAME: SOURCE MODEL $\rightarrow$ TARGET MODEL

Summary description of the transformation from the *SOURCE MODEL* instance to the *TARGET MODEL* instance. The *SYMBOLIC NAME* is used for referencing in Figures and textual descriptions.

Input: Description of the input model instance (*SOURCE MODEL*)

Output: Description of the output model instance (*TARGET MODEL*)

Dependencies and Requirements: Description of dependencies on other transformation steps and requirements towards the model instances.

Pre-processing: Description of any necessary pre-processing during the transformation before any of the transformation rules are executed.

SOURCE ELEMENT $\rightarrow$ TARGET ELEMENT Description of the transformation of a *SOURCE ELEMENT* from the *SOURCE MODEL* into a *TARGET ELEMENT* of the *TARGET MODEL*. There may be any number of element transformation descriptions. The order of the element transformation descriptions does not reflect the execution order. The execution order is determined by the ETL runtime environment.

OPERATION: Description of an imperative operation executed on the whole model instance. There may be any number of operations. The operations

[33]See `http://www.eclipse.org/epsilon/doc/eol/`.
[34]See `http://www.eclipse.org/epsilon/doc/etl/`.

are generally listed in the order of intended or required execution, unless specified otherwise.

Post-processing: Description of any necessary post-processing during the transformation after all of the transformation rules are executed.

The placeholders *SOURCE MODEL*, *TARGET MODEL*, *SOURCE ELEMENT*, and *TARGET ELEMENT* are replaced with concrete model instances and elements in the transformation descriptions, and the placeholder *OPERATION* is replaced by a name for a concrete operation. The descriptions according to this template shall provide a more accessible summary of the transformation activities without requiring prior knowledge of the implementation technology. Snippets of relevant EOL and ETL code may be provided where necessary. The complete detailed transformation specifications can be found in Appendix A.2.

While Figure 5.2 presents a rather simplified view on the transformation activities during the facts transformation step, in practice the transformation workflow is a bit more complicated, due to dependencies between the different transformations as well as intermediate transformations added for performance or convenience reasons. The main part of the transformation workflow is shown in Figure 5.16. The transformation activities (shown in gray) are ordered according to their general temporal precedence from left to right, where certain transformations such as *MG2CFA* and *FAMIX2DECENT* may be performed in parallel as they are independent from each other and operate on different target models. Additional transformations for enriching the *DECENT* model with collaboration, experience, and temporal characteristics are shown in Figure 5.17, where the list of transformations can be extended beyond the listed ones. The only constraint is that the *DELTA2DECENT* transformation is executed before the *DECENT* model is used in an assessment task in order update the *Deltas* for any new *Attributes*.

5.4.4.1. VCS Repository Facts

The VCS repository model described by the *MG* meta-model serves as the backbone both for the target assessment model described by the *DECENT* meta-model and for the intermediate model used for the realisation of the cause-fix analysis described in Chapter 3. Hence, the VCS repository facts are involved in two transformations — the transformation of the parts of the *MG* instance relevant to the assessment task into a new *DECENT* instance (*MG2DECENT*) and the transformation of the parts of the *MG* instance relevant to the cause-fix analysis into a new *CFA* instance ((*MG2CFA*)). Furthermore, the revision hierarchy information from the *DAG* instance is also integrated in the *DECENT* at a later point (*DAG2DECENT*).

MG2DECENT: MG $\rightarrow$ DECENT

The transformation from the *MG* to the *DECENT* model takes an existing *MG*

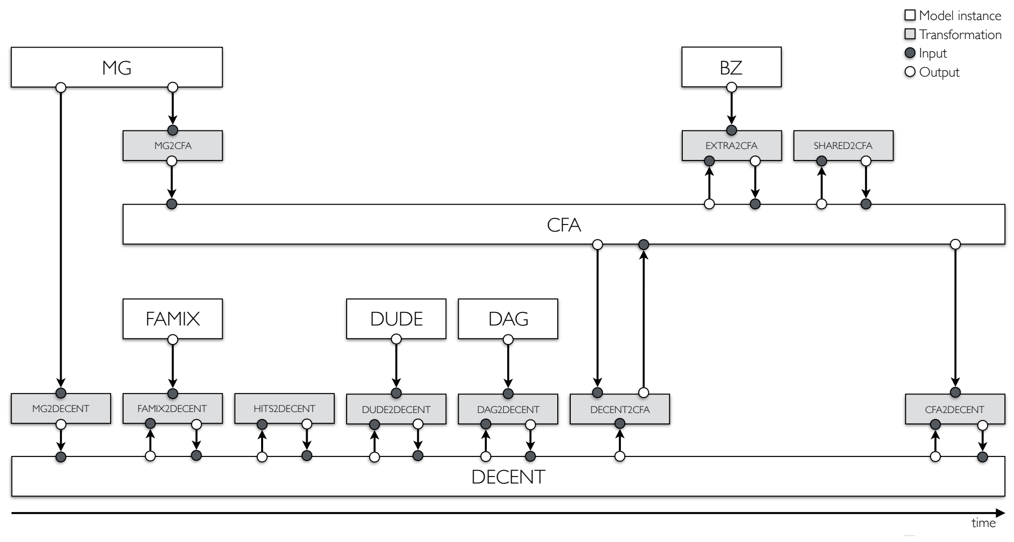


Figure 5.16.: Transformation workflow (part 1)

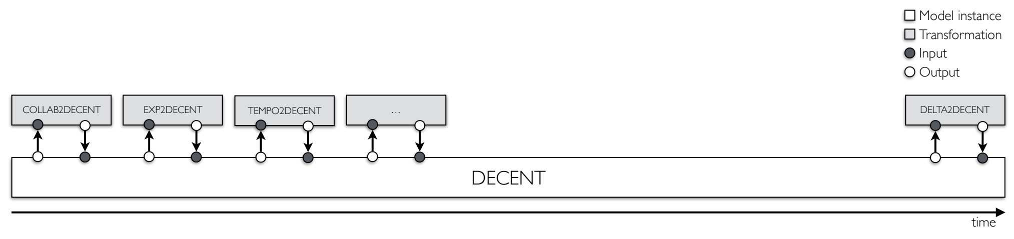


Figure 5.17.: Transformation workflow (part 2)

model instance and creates a new *DECENT* model instance containing relevant facts from the *MG* model instance.

Input: An existing *MG* model instance.

Output: A new *DECENT* model instance.

Dependencies and Requirements: The *MG* model instance shall be pre-processed to include normalised *Hunks* descriptions.

Pre-processing: Initialise the necessary *Attribute* element definitions. Initialise the *"Spatial"*, *"Change"*, and *"File" Dimensions*. Initialise the list of selected *Branches*.

Model $\rightarrow$ Model: Transform the top-level *Model* elements from the *MG* instance into top-level *Model* elements of the *DECENT* instance. Add all *Attribute* elements to the *Model*.

People → Developer: Transform *People* elements from the *MG* instance into *Developer* elements in the *DECENT* instance.

Revision → Developer State: Transform *Revision* elements from the *MG* instance into *DeveloperState* elements in the *DECENT* instance. Assign the *ID* property of the *DeveloperState* to the *commit_id* property of the *Revision*. Add a *Value* for the *"Timestamp"* attribute containing the timestamp derived from the *author_date* property of the *Revision*.

Branch → Artifact: Transform *Branch* elements from the *MG* instance into *Artifact* elements of the branch *ArtifactType*.

File → Artifact: Transform *File* elements from the *MG* instance into *Artifact* elements of the corresponding *ArtifactType* in the *DECENT* instance, creating missing *ArtifactType* elements as necessary. Reconstruct the respective *Artifact* hierarchy based on the *File* hierarchy reflected in the *FileLink* elements in the *MG* instance.

Action → Artifact State: Transform *Action* elements from the *MG* instance into *ArtifactState* elements in the *DECENT* instance. The resulting *ArtifactState* elements are contained in the *Artifact* elements corresponding to the *File* elements associated with each *Action* element. The *ID* property of the *ArtifactState* element is assigned to the *commit_id* property of the *Revision* associated with each *Action* element. *Value* elements for the *"FilePath"*, *"BranchName"*, *"FileSize"*, *"LOC"*, and *"AggregateFragmentCount"* attributes are added to the resulting *ArtifactState* based on corresponding properties from the *MG* instance, derived from the *Action*, the associated *Branch*, the *Content* and *LineBlames* associated with the *Revision* and *File* related to the *Action*, respectively. The previous *ArtifactState* for the containing *Artifact* is determined and associated with the resulting *ArtifactState*. *Activity* elements are created to link the resulting *ArtifactState* to the respective previous *ArtifactState* and to the corresponding *DeveloperState*. *Value* elements for the characteristics *"ChangedFragmentCount"*, *"LinesAdded"*, *"LinesRemoved"*, and *"CommitMessage"* related to the *Activity* itself are created based on corresponding properties derived from the number of *Hunks* and the properties of the *PatchLines* associated with the *Revision* and *File* related to the *Action*, as well as the *"message"* property of the associated *Revision*, respectively. Finally, *Value* elements for the *Spatial Characteristics* are added to the resulting *ArtifactState*.

Post-processing: Add a *Value* for the *"Tags"* attribute to each *ArtifactState* element in the *DECENT* instance. The *Tags* attribute contains the names of all the *Tag* model elements from the *MG* instance that were assigned to revisions spanning the time frame between the time of the activity leading to the *ArtifactState* and the time of the next activity performed on the *ArtifactState*. Add a *Value* for the *"TagCount"* attribute to each *ArtifactState* element in the *DECENT* instance containing the number of elements in the *"Tags"* value.

MG2CFA: MG → CFA

The transformation from the *MG* to the *CFA* model takes an existing *MG* model instance and creates a new *CFA* model instance containing cause-fix relationships at the project and file levels of granularity derived from the *MG* model instance.

Input: An existing *MG* model instance.

Output: A newly created *CFA* model instance.

Dependencies and Requirements: None.

Pre-processing: Initialise the necessary *Attribute* element definitions for the different weights, including *"RemovedWeight"*, *"TotalWeight"*, and *"AverageWeight"*. Initialise the *"Default"* and *"BugFix" Factor* elements. Initialise the *"Inherit" Strategy*. Assign the *"Inherit" Strategy* to both *Factor* elements.

Model → Model: Transform the top-level *Model* elements from the *MG* instance into top-level *Model* elements of the *CFA* instance. Add all *Attribute* and *Factor* elements to the *Model*.

Revision → Project State: Transform *Revision* elements from the *MG* instance into *ProjectState* elements. Assign the values of the *"name"* and *"ID"* properties of the *ProjectState* to the values of the *"rev"* and *"commit_id"* properties of the *Revision*, respectively. Add the *FactorEntry* elements for the *"Default"* and *"BugFix" Factors*. Add *AttributeEntry* elements for the *"RemovedWeight" Attribute* for each *Factor*, where the value for the *"Default" Factor* is always 1, and the value for the *"BugFix" Factor* is assigned to the value of the *"is_bug_fix"* property of the *Revision*. Determine the causing *ProjectStates* by navigating the *Hunks* related to the *Revision* and the corresponding *HunkBlames* related to each *Hunk*.

Action → File State: Transform *Action* elements from the *MG* instance into *FileState* elements. Assign the values of the *"name"* and *"ID"* properties of the *FileState* to the values of the *"current_file_path"* property of the *Action* and the *"commit_id"* property of the *Revision* associated with the *Action*, respectively. Determine the containing *ProjectState* and add the resulting *FileState* to it. Add the *FactorEntry* elements for the *"Default"* and *"BugFix" Factors*. Add *AttributeEntry* elements for the *"RemovedWeight" Attribute* for each *Factor*, where the value for the *"Default" Factor* is always 1, and the value for the *"BugFix" Factor* is assigned to the value of the *"is_bug_fix"* property of the *Revision* associated with the *Action* (*inherit* strategy). Determine the causing *FileStates* by navigating the *Hunks* related to the *Revision* associated with the *Action* and the corresponding *HunkBlames* related to each *Hunk*.

Post-processing: Calculate the values for the *"TotalWeight" Attribute* for all *States*. Calculate the values for the *"AverageWeight" Attribute* for all *States*.

DAG2DECENT: DAG → DECENT

The transformation from the *DAG* to the *DECENT* model takes an existing *DAG* model instance and refines an existing *DECENT* model instance for the same project.

Input: An existing *DAG* model instance, an existing *DECENT* model instance, both shall be related to the same project.

Output: A refined *DECENT* model instance.

Dependencies and Requirements: Both the *DAG* and the *DECENT* model instances shall be related to the same project.

Check Correct Artifact State Sequence: Check the state sequences for each *Artifact* against to the *DAG Graph* and fix mismatching state sequences where necessary.

Assign Developer State Sequence: Assign the previous *DeveloperState* for each *DeveloperState* of each *Developer* based on the *DAG Graph*.

In the *MG2CFA* transformation we relied on information already provided within the *MG* model to determine the causes for each state. As discussed in Chapter 3, other more sophisticated approaches may be used instead to refine the resulting cause-fix relationships. In addition, we only assigned weights for two *Factors* and used the *inherit* strategy by default for assigning *RemovedWeights* to *FileStates* so far. In subsequent transformations, we will discuss adding other *Factors* and applying different strategies for weights distribution between the different levels of granularity. In the descriptions of the transformations from the *MG* model, we assumed that each *Model* element in the *MG* model represents a single project. This simplification may not necessarily apply in practice, thus some interpolation may be necessary. The *DAG2DECENT* transformation checks for the correct *ArtifactState* sequence, hence it shall be executed after all *ArtifactStates* have been added.

5.4.4.2. Static Code Analysis Facts

The *FAMIX* model instances containing the extracted static code analysis facts for each revision are used to enrich and refine the target assessment model described by the

DECENT meta-model with information about artifacts at the logical level of granularity. Each *FAMIX* model instance is transformed individually into the target *DECENT* model instance. After that, facts from the *DUDE* model instance containing related to detected duplicates are also transformed into target assessment model by creating *Value* elements for the duplicate-related attributes within the corresponding *ArtifactState* elements.

FAMIX2DECENT: FAMIX → DECENT

The transformation from the *FAMIX* to the *DECENT* model takes a set of existing *FAMIX* model instances for individual revisions and refines an existing *DECENT* model instance for the same project. Transformations for *Class*, *Method*, *Module*, and *Function* elements are nearly identical, therefore they are summarised as *LOGICAL* elements.

Input: A set of existing *FAMIX* model instances, an existing *DECENT* model instance, both shall be related to the same project.

Output: A refined *DECENT* model instance.

Dependencies and Requirements: Both the *FAMIX* and the *DECENT* model instances shall be related to the same project. The *FAMIX* model instance shall include an indication of the revision for which it was extracted, allowing model elements to be mapped to corresponding *ArtifactStates* at the file level of granularity in the *DECENT* model.

Pre-processing: Identify *ArtifactStates* at the file level of granularity that correspond to the revision for which the *FAMIX* model instance was produced. Initialise the *"Logical" Dimension*.

LOGICAL → Artifact State Transform *LOGICAL* elements to *ArtifactState* elements. Identify the corresponding parent *ArtifactState*. Ignore *LOGICAL* elements for which the containing *Artifact* element does not have a corresponding *ArtifactState* for the revision for which the *FAMIX* model was produced. Create necessary *Artifact* elements contained in the corresponding parent *Artifact* element where necessary. Set the previous *ArtifactState* if it exists. Add *Value* elements for the *"FileAnchor"*, *"StartLine"*, and *"EndLine" Attributes* of the *ArtifactState* for the corresponding properties of the *FileAnchor* element associated with the *LOGICAL* element in order to record the location of the *LOGICAL* element. Assign the *"Spatial" Dimension* to these *Attributes*. Add *Value* elements for each property of type *Real* from th e *LOGICAL* element to the resulting *ArtifactState* element, creating the necessary *Attributes* and assigning the *"Logical" Dimension* to them. Create an *Activity* element to link the *ArtifactState* to the previous *ArtifactState* and to the corresponding *DeveloperState*.

Post-processing: None.

DUDE2DECENT: DUDE → DECENT

The transformation from the *DUDE* to the *DECENT* model takes an existing *DUDE* model instance and refines an existing *DECENT* model instance with clone-related information at the file and logical levels of granularity.

Input: An existing *DUDE* model instance, an existing *DECENT* model instance, both shall be related to the same project.

Output: A refined *DECENT* model instance.

Dependencies and Requirements: Both the *DUDE* and the *DECENT* model instances shall be related to the same project.

Pre-processing: Initialise the necessary *Attribute* element definitions and the *"Clone" Dimension*.

Assign Clone Information to States: Assign cloned lines and the number of cloned fragments in each *ArtifactState*, based on the information from the *Run* executed on the revision corresponding to the *ArtifactState*.

Post-processing: None.

HITS2DECENT: DECENT → DECENT

The transformation enriches an existing *DECENT* model with information on which parts *Artifacts* at the logical level of granularity were modified in each *ArtifactState*.

Input: An existing *DECENT* model instance already containing information for *Artifacts* at the logical level of granularity (*FAMIX2DECENT* has been executed).

Output: A refined *DECENT* model instance.

Dependencies and Requirements: The *FAMIX2DECENT* transformation shall be executed before the *HITS2DECENT* transformation.

Pre-processing: Initialise the necessary *Attribute* element definitions and the *"Hits" Dimension*.

Assign Modified Lines Information to States: Assign modified lines and *Spatial Characteristics* to *ArtifactStates* at the logical level of granularity based on the *Spatial Characteristics* of the corresponding parent *ArtifactState* at the file level of granularity. This is done by comparing the *"StartLine"* and *"EndLine" Values* of the *ArtifactState* at the logical level of granularity to the *"LinesPost"* values for the corresponding parent *ArtifactState*. In addition, a *Value* for the *"VariancePostLines"* attribute describing the variance of the modified lines within the artifact is also added to the *ArtifactState* at the logical level of granularity.

Post-processing: None.

In order to add cause-fix relationships at the logical level of granularity to the *CFA* model instance, we use the *DECENT* model instance already containing the relevant information and relationships between *Artifact* and *ArtifactStates* at the file and logical levels of granularity.

DECENT2CFA: DECENT → CFA

The transformation enriches an existing *CFA* model with information at the logical level of granularity derived from a *DECENT* model instance for the same project.

Input: An existing *CFA* instance containing cause-fix relationships at the project and file levels of granularity, an existing *DECENT* model instance already containing information for *Artifacts* at the logical level of granularity (*FAMIX2DECENT* has been executed) and which *ArtifactStates* at the logical level of granularity have been modified (*HITS2DECENT* has been executed).

Output: A refined *CFA* model instance containing information about cause-fix relationships at the logical level of granularity.

Dependencies and Requirements: The *FAMIX2DECENT* transformation and the *HITS2DECENT* transformation shall be executed before the *DECENT2CFA* transformation.

Pre-processing: None.

Artifact State → Logical State: Transform modified *Artifact State* elements for *Artifacts* of logical *ArtifactTypes* from the *DECENT* instance into *LogicalState* elements. Assign the values of the *"name"* and *"ID"* properties of the *LogicalState* to the values of the *"name"* property of the corresponding *Artifact* associated with the *ArtifactState* and the *"ID"* property of the *Artifact*, respectively. Determine the containing *ProjectState* and add the resulting *LogicalState* to it. Add the *FactorEntry* elements for the *"Default"* and *"BugFix" Factors*. Add *AttributeEntry* elements for the *"RemovedWeight" Attribute* for each *Factor*, where the value for the *"Default" Factor* is always 1, and the value for the *"BugFix" Factor* is assigned to the value of the corresponding *Attribute* of the *ProjectState*. Determine the *FileState* corresponding to the parent *Artifact* for the *Artifact* associated with the *ArtifactState*. Determine the causing *LogicalStates*.

Post-processing: Reset the values for the *"'TotalWeight' Attribute* in all *States*. Calculate the values for the *"TotalWeight" Attribute* for all *States*. Calculate the values for the *"AverageWeight" Attribute* for all *States*.

5.4.4.3. ITS Repository Facts

The *BZ* model instance containing the facts extracted from the ITS are used to enrich *CFA* model instance with additional *Factors* and associated weights based on the relationships between *ProjectStates* and reported *Issues*. We consider the following *Factors* for which the value of the *"RemovedWeight" Attribute* can serve as an indication of the importance or impact of a fixing state and hence also indicate that the corresponding causes for the fixing state also have a potentially high impact and shall be treated with more caution:

Issue Count: Number of *Issues* related to a *State*.

Comments Per Issue: Average number of *Comments* per related *Issue*.

Users Per Issue: Average number of distinct users submitting *Comments* per related *Issue*.

Issue Importance: Average importance ranking per related *Issue*. The importance classification may vary across projects, hence a project-specific mapping from the ordinal scale of importance classifications to the interval $[0,1]$ is necessary.

Additional *Factors* related to ITS facts may be added at a later point as well. In addition to the *Factors* derived from the ITS, we also add several other *Factors* based on regular expressions evaluated against the description of a *Revision* as indicated in its *"message"* property:

Refactoring: Whether or not a fixing state includes a refactoring, based on the regular expression `.+(factored|factoring).*`.

Fix: A softer version of the *"BugFix" Factor* (which is derived from the classification already present in the *MG* model). The *"Fix" Factor* is based on the regular expression `.*(fix|bug|bug:).*`.

EXTRA2CFA: BZ → CFA

The transformation from the *BZ* to the *CFA* model takes an existing *BZ* model instance and refines an existing *CFA* model instance with ITS-related factors and weights.

Input: An existing *BZ* model instance, an existing *CFA* model instance, both shall be related to the same project.

Output: A refined *CFA* model instance.

Dependencies and Requirements: Both the *BZ* and the *CFA* model instances shall be related to the same project.

Pre-processing: Initialise the necessary *Factor* elements for the *“IssueCount”*, *“CommentsPerIssue”*, *“UsersPerIssue”*, *“IssueImportance”*, *“Refactoring”*, and *“Fix”* factors. Assign the *“Inherit” Strategy* to all new *Factor* elements. Map *BZIssues* to *ProjectStates* based on multiple indicators, such as references to the *“name”* of the *ProjectState* (derived from the *Revision* in the *MG* model instance during the *MG2CFA* transformation).

Add Factors and Removed Weights to States: Add the *FactorEntry* elements for the *“IssueCount”*, *“CommentsPerIssue”*, *“UsersPerIssue”*, *“IssueImportance”*, *“Refactoring”*, and *“Fix” Factors* to each *State*. Add *AttributeEntry* elements for the *“RemovedWeight" Attribute* for each *Factor*. *FileStates* and *LogicalStates* inherit the values of the *“RemovedWeight” Attribute* for each *FactorEntry*.

Post-processing: Calculate the values for the *“TotalWeight” Attribute* for the newly added *Factors* for all *States*. Calculate the values for the *“AverageWeight” Attribute* for the newly added *Factors* for all *States*.

So far the values for the *“RemovedWeight” Attribute* for all the *Factors* added to the *CFA* model were simply copied across the different levels of granularity (*inherit* strategy). Next, we discuss the application of the different strategies described in Section 3.5 to the *CFA* model.

SHARED2CFA: CFA → CFA

The transformation refines an existing *CFA* model with redistributed weight according to the different strategies discussed in Section 3.5. It creates additional *Factors* for each existing *Factor* and each applied strategy so that weighting resulting from the different strategies can be compared and combined.

Input: An existing *CFA* model instance already containing information for *Artifacts* at the file and logical levels of granularity (*DECENT2CFA* has been executed).

Output: A refined *CFA* model instance containing additional *Factors* for each *Factor* and each applied strategy.

Dependencies and Requirements: The *DECENT2CFA* transformation shall be executed before the *SHARED2CFA* transformation. If the *EXTRA2CFA* transformation is executed, the *SHARED2CFA* shall be executed (again) after it.

Pre-processing: Initialise the additional *Strategy* elements for the *“Shared”*, *“Type”*, *“Size”*, and *“Churn”* strategies. Initialise the additional *Factor* elements for each existing *Factor* associated with the *“Inherit”* strategy and each new *Strategy* and associate them with the corresponding *Strategy*.

Apply Strategies to Factors: Add the new *FactorEntry* elements for all *Factors* associated with a *Strategy* different than the default *"Inherit" Strategy* to each *State*. Add *AttributeEntry* elements for the *"RemovedWeight" Attribute* for each *Factor*. Assign the values of the *"RemovedWeight" Attribute* for each *Factor* according to the *Strategy* for the corresponding *Factor*.

Post-processing: Calculate the values for the *"TotalWeight" Attribute* for the newly added *Factors* for all *States*. Calculate the values for the *"AverageWeight" Attribute* for the newly added *Factors* for all *States*.

After obtaining a comprehensive *CFA* model instance containing weights related to different properties, distributed across different levels of granularity according to different strategies, we need to integrate the weighting information back into the *DECENT* model instance.

CFA2DECENT: CFA $\rightarrow$ DECENT

The transformation refines an existing *DECENT* model with weighting information from an existing *CFA* model, which is assigned to *ArtifactStates* indicating their likelihood for causing different kinds for events of interest.

Input: An existing *CFA* model instance and an existing *DECENT* model instance, both related to the same project.

Output: A refined *DECENT* model instance containing weighting information indicating the likelihood of *ArtifactStates* for causing different kinds of events of interest.

Dependencies and Requirements: The *CFA2DECENT* transformation shall be executed after any changes have been made to the *CFA* model (e.g. applying a different *Strategy* or adding a new *Factor*).

Pre-processing: Initialise the *"Cause-Fix" Dimension*. Initialise *Attributes* in the *DECENT* model for each combination of *Factors* and *Attributes* from the *CFA* model. Initialise additional *Attributes* for *"CarriedWeight"* for each of the newly initialised *Attributes*. Initialise *Attributes* for the number of caused future *ArtifactStates* (*"CausesCount"*) and the number of past *ArtifactStates* causing the *ArtifactState* (*"FixesCount"*).

Add Weights to Artifact States: Map *FileStates* and *LogicalStates* to corresponding *ArtifactStates*. Transform the values for all *AttributeEntries* into corresponding *Values* of the mapped *ArtifactState*. Add *Values* for the *"CausesCount"* and *"FixesCount" Attributes* to the mapped *ArtifactState*.

Add Fixes and Causes Counts to Developer States: Map *ProjectStates* to corresponding *DeveloperStates*. Add *Values* for the *"CausesCount"* and *"FixesCount" Attributes* to the mapped *DeveloperState*.

Add Carried Weights: Add *Values* for the *"CarriedWeight" Attributes* based on the sum of *"TotalWeights"* for the corresponding *Factor* accumulated over time, where *"RemovedWeights"* derived from the corresponding *Factor* are subtracted in each *ArtifactState*.

Add Temporal Characteristics: Calculate temporal distances between *Causes* and *Fixes* for each *Factor* at each level of granularity. Add the minimum, maximum, mean, and standard deviation characteristics to the *DECENT* model, which can be used to determine the confidence windows for the weights.

Post-processing: None.

5.4.4.4. Derived Facts and Deltas

Beyond the integration of facts from the different facts models in to the assessment model, we also derive additional facts based on the characteristics and structure of the assessment model. These include collaboration-related facts, experience-related facts, and temporal facts. Finally, we also calculate the *Deltas* between the source and target *ArtifactStates* for each *Activity*.

COLLAB2DECENT: DECENT → DECENT

The transformation refines an existing *DECENT* model with the collaboration-related facts described in Section 4.2.5.

Input: An existing *DECENT* model instance.

Output: A refined *DECENT* model instance containing *Values* for collaboration-related *Attributes*.

Dependencies and Requirements: If the *FAMIX2DECENT* transformation has been executed, the *HIT2DECENT* transformation shall be executed as well.

Pre-processing: Initialise the *"Collaboration" Dimension*. Initialise necessary collaboration-related *Attributes*.

Add Collaboration Characteristics to Artifact States: Identify collaborating *Developers* up to the point in time a given *ArtifactState* was created. Calculate and add the *Values* for the collaboration-related *Attributes*.

Add Collaboration Characteristics to Developer States: Identify *Artifacts* on which a given *Developer* has collaborated up to the point in time of the *DeveloperState*. Based on these, identify collaborating *Developers* up to the point in time of the *DeveloperState*. Calculate and add the *Values* for the collaboration-related *Attributes* across each *ArtifactType* individually.

Post-processing: None.

EXP2DECENT: DECENT → DECENT

The transformation refines an existing *DECENT* model with the experience-related facts described in Section 4.2.4.

Input: An existing *DECENT* model instance.

Output: A refined *DECENT* model instance containing *Values* for experience-related *Attributes*.

Dependencies and Requirements: If the *FAMIX2DECENT* transformation has been executed, the *HIT2DECENT* transformation shall be executed as well.

Pre-processing: Initialise the *"Experience" Dimension*. Initialise necessary experience-related *Attributes*.

Add Experience Characteristics to Artifact States: Calculate and add the *Values* for the experience-related *Attributes* for each *ArtifactState*.

Add Experience Characteristics to Developer States: Calculate and add the *Values* for the experience-related *Attributes* for each *DeveloperState* across each *ArtifactType* individually..

Post-processing: None.

TEMPO2DECENT: DECENT → DECENT

The transformation refines an existing *DECENT* model with the temporal facts described in Section 4.2.3.

Input: An existing *DECENT* model instance.

Output: A refined *DECENT* model instance containing *Values* for temporal *Attributes*.

Dependencies and Requirements: If the *FAMIX2DECENT* transformation has been executed, the *HIT2DECENT* transformation shall be executed as well.

Pre-processing: Initialise the *"Temporal" Dimension*. Initialise necessary temporal *Attributes*.

Add Temporal Characteristics to Artifact States: Calculate and add the *Values* for the temporal *Attributes* for each *ArtifactState*.

Add Temporal Characteristics to Developer States: Calculate and add the *Values* for the temporal *Attributes* for each *DeveloperState* across each *ArtifactType* individually..

Post-processing: None.

DELTA2DECENT: DECENT → DECENT

The transformation refines an existing *DECENT* model with *Deltas* between the source and target *ArtifactStates* for each *Activity*.

Input: An existing *DECENT* model instance.

Output: A refined *DECENT* model instance containing *Deltas*.

Dependencies and Requirements: The *DELTA2DECENT* transformation shall be executed after any changes have been made to the *DECENT* model resulting in *Values* for new *Attribute* or new *ArtifactStates* and *Activities*.

Pre-processing: None.

Add Deltas to Activities: Calculate *Deltas* for all *IntegerValues* and *RealValues* of a modified *Artifact State* and add them to the corresponding *Activity* from which the *ArtifactState* resulted.

Post-processing: None.

5.4.5. Assessment Transformation

After obtaining a comprehensive *DECENT* model instance containing multi-faceted information describing the behaviour of developers, we proceed to make use of this information in various assessment applications. We query the *DECENT* model instance to obtain different views on the information, such as the behaviour of a given developer at a particular level of granularity as defined by the activities of the developer at that level of granularity as well as the context in which they occurred and their outcome.

Assessment applications, such as machine learning approaches for defect prediction, typically provide binary classification and hence also require training data with corresponding binary classifications. A binary indication (*true* or *false*) of whether an activity contributed to causing an events of interest, such as a bug fix or a refactoring, can be computed based on the value for the associated weight-related attributes derived from the *CFA* model, such as the average weight for the corresponding factor. The threshold for determining the binary value can be based on the distribution of the average weights. In addition, a confidence indicator (*high* or *low*) can be added to note whether the binary indicator can be trusted, e.g. if it is very close to the threshold.

On the other hand, a visualisation may use color shading to indicate the likelihood that a state of an artifact can be considered a cause of an even of interest. Thus, depending to the target assessment application, further characteristics may need to be calculated during the transformation into the assessment assets expected by the assessment application. For certain applications, part of the characteristics may also need to be hidden. Contemplating the machine learning approach again, if we calculate a binary classification based on a threshold applied to the weight-related attributes, then the basis for that classification shall not be part of the exported data. Otherwise, the machine learning application is quickly going to learn that indeed a given threshold on the weight-related attribute (likely very close to the one used for deriving the binary indicators in the first place) is the best way to partition the data.

Depending on the target assessment applications it can be beneficial to address the needs of each individual application by means of transformations to the corresponding types of target assets, as shown in Figure 5.2. As a feasibility study, for the realisation of the patterns and applications discussed Section 4.3 we decided to consolidate our efforts around a single integrated platform and single type of assessment assets for simplicity.

For data mining applications, including determining the importance of characteristics, clustering similar activities, and predicting causes for events of interest, we rely on *Weka* [64]. *Weka* provides rich facilities for various machine learning tasks. It supports input in CSV and *Attribute-Relation File Format* (ARFF)[35] format. Data from the *DECENT* model instance can be exported in these formats by means of M2T transformation. However, once the data is in that format it is "dumbed down", thus no longer accessible at the model level and it does not contain any additional meta-data. For more convenient pre-processing of the application-specific data at the model level, we designed an intermediate model representation that is closely related to the ARFF format used by *Weka*. This model representation is described by the *ARFFx* meta-model shown in Figure 5.18. The *ARFFx* meta-model includes concepts related to the input for *Weka*, including *Instances* containing *Values* of *Attributes*, which have a *Type* and a *Dimension*. The *Dimension* is a convenience extension to the original ARFF format, which is based on the concept of the same name in the *DECENT* meta-model. It is used for categorising and filtering *Attributes* so that the same *ARFFx* model can be used to derive multiple concrete views, such as a data set including only collaboration or temporal characteristics. The *ARFFx* model also includes the *MetaData* concept which is used for describing the contents of the concrete model instance and any pre-processing steps that may have been applied to it, as well as additional information, such as relevant thresholds.

The *ARFFx* model instances are populated by querying and transforming the *DECENT* model.

DECENT2ARFFx: DECENT → ARFFx

The transformation transforms an existing *DECENT* model instance into a new *ARFFx* model instance, containing individual *Models* for each *Developer* at each level of granularity, as well as for all *Developers*.

Input: An existing *DECENT* model instance.

Output: A new *ARFFx* model instance containing multiple *Models*.

Dependencies and Requirements: The *DECENT2CFA* transformation shall be executed before the *DECENT2ARFFx* transformation.

Pre-processing: None.

[35]See `http://www.cs.waikato.ac.nz/ml/weka/arff.html`.

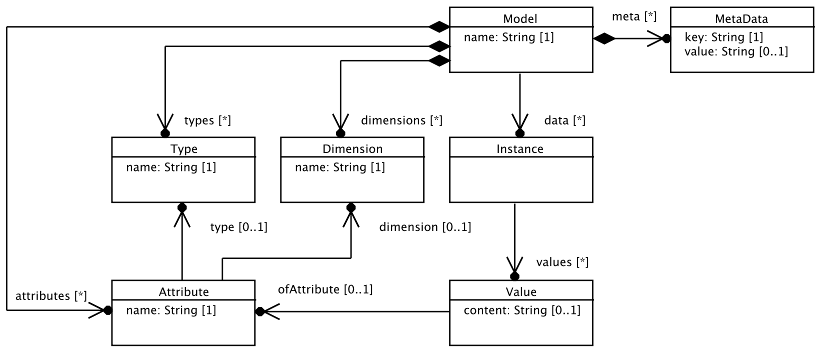


Figure 5.18.: *ARFFx* meta-model for the structure of assessment assets used in *Weka*

Export Developer Behaviours: Create new *Model* element for each *Developer*, at each level of granularity, containing the behaviour of the *Developer* at that level of granularity. Each *Instance* of the *Model* contains *Values* for *Attributes* based on the source *ArtifactState* of an *Activity*, the target *ArtifactState* of the *Activity*, the associated *DeveloperState* for the *Activity*, and *Values* and *Deltas* of the *Activity* itself.

Export Baseline: Create new *Model* element for each level of granularity, containing the behaviour of all *Developers*, described in an identical way as the individual *Developer* behaviours.

Post-processing: None.

Once the *ARFFx* model instances are populated, they are processed further and exported by M2T transformation to individual CSV assets which are used for experiments with *Weka*. The processing involves assigning binary classifications based on different weight-related *Attributes* and a given threshold (based on the mean value for the *Attribute*), as well as confidence labels indicating the confidence in the assigned binary classification. The attributes on which the classifications are based, as well as related attributes can be filtered during the transformation to CSV assets. *Models* for individual *Developers* at a given level of granularity containing a small number of *Instances* (below a given threshold) can be grouped together to describe a generic behavior of *"small contributors"* at that level of granularity. Similarly, *Models* for individual *Developers* containing a large number of *Instances* can be grouped together to describe a generic behavior of *"big contributors"*. Finally, using temporal characteristics related to the distances between events of interest and their causes can be used to determine confidence windows for activities across the different factors. Since the information related to the causes for events of interest depends on knowing the events of interest, the information for the last activities for a recorded period of time is inherently incomplete. Based on the average time span $\overline{\mathrm{d}}_{cause \to fix}$ between an event of interest and its cause, we can infer the point in time $t_{confident} = t_{max} - \overline{\mathrm{d}}_{cause \to fix}$ after which we cannot be certain that the information regarding the causes for events of interest is reliable. For additional confidence, we may also consider the standard deviation of the time spans so that the point in time after which we cannot be certain that the information regarding the causes for events of interest is reliable becomes $t_{extraconfident} = t_{max} - (\overline{\mathrm{d}}_{cause \to fix} + \sigma_{cause \to fix})$. The confidence windows may vary for different factors.

For visual inspection, including mapping developer activities in time, visualising developer ranks and collaboration networks, as well as outcomes from the data mining applications, we rely on the *Processing*[36] platform. *Processing* is a flexible platform for design and prototyping of large-scale motion graphics and complex data visualisation in the context of the visual arts. Since we integrate outcomes from the data mining applications, we decided to use the facilities provided by *Weka* for managing the data for the visualisation in order to rely on a unified data access layer and streamline the overall process. Hence, we can reuse the assessment assets produced for the data mining applications.

5.4.6. Assessment Application

The data mining applications for determining the importance of characteristics, clustering similar activities, and predicting causes for events of interest are realised by means of a customised interface to *Weka*, based on CrossPare[37] [74]. *CrossPare* is a tool for executing cross-project defect-prediction experiments and benchmarks. The customised interface based on *CrossPare* provides facilities for automating the assessment applications discussed in Section 4.3. By means of a comprehensive set of parameters, different modes of operation can be selected and configured according to the task at hand. The modes include:

- **rank-attributes** for determining the importance of characteristics over time
- **predict** for predicting causes for events of interest, supporting different kinds of machine learning algorithms
- **cluster** for clustering similar activities by means of the *kmeans* algorithm, inspecting the defining characteristics of the clusters, as well as predicting causes for events of interest within the clusters
- **developer-centric** for predicting causes for events of interest based on predictive models for individual developers or groups of developers

[36]See `https://www.processing.org`.

[37]See `https://crosspare.informatik.uni-goettingen.de/trac`.

- **developer-crossover** for predicting causes for events of interest based on predictive models transferred across individual developers, both within the same project and across different projects
- **cluster-crossover** for predicting causes for events of interest based on predictive models for groups of similar activities transferred across individual developers or whole projects
- **artifact-centric** for predicting causes for events of interest based on predictive models for individual artifacts or groups of artifacts

The different modes can be configured to apply various pre-processing steps, such as normalising the values of the different characteristics, sorting the activities, weighting to offset imbalanced data for predictive modelling, applying sub-sampling, as well as selecting subsets of characteristics based on their importance or other criteria. Furthermore, the parameters can be used to determine the data partitioning for the training and testing of the predictive models, as well as the resolution for the ranking of the characteristics. The output is typically in a structured textual format, including measurements of success indicating how good a predictive model performed against a test set, as well as an optional description of the circumstances determining the outcome of activities by a given developer or group of developers.

In order to support the visual inspection and visualise different patterns as well as outcomes from the data mining applications, we implemented custom viewers based on the *Processing* platform. While off-the-shelf tools and libraries for visualisation can provide a quick and easy way to get a first glimpse into the data by means of common visual representations, there are often limitations when it comes to advanced visualisation and interaction capabilities. The *Processing* platform provides full control and flexibility over everything that is drawn as well as how a user can interact with it, while still hiding most of the low-level complexity. We created the following viewers:

- **activity-viewer** for mapping developers activities on artifacts over time, with additional overlays for visualising prediction results and cluster assignments
- **ranking-viewer** for displaying developer ranks over time
- **front-ranking-viewer** for displaying developer rankings at the project front
- **attribute-ranking-viewer** for displaying the importance of characteristics over time
- **spider-viewer** for displaying the importance of characteristics at a given point in time in the form of a Kiviat diagram
- **collaboration-viewer** for displaying and laying out the collaboration networks

To aid the navigation and exploration of data, we created additional GUI viewers by means of the *Standard Widget Toolkit* (SWT)[38] which glue the various assessment and

[38]See `https://www.eclipse.org/swt/`.

visualisation functionalities together.

In a related project [80, 82], agent-based simulation applications making use of the developer centric assessment have been realised by means of the *Repast Symphony*[39]. They are used for simulating software evolution and answering research questions related to system growth, software changes, as well as developer collaboration and involvement.

5.5. Related Work

Existing work often stresses that data extraction and preparation in the context of MSR is a complex and time-intensive task [66, 72, 151]. A frequent critique is that there is no common vocabulary of terms or data representation techniques across different works on MSR. Instead, assessment-specific ad-hoc data representations are used (compare, for example [60, 138, 147, 168, 193]). Aside from the lost research efficiency, the multitude of data extraction and representation approaches makes it hard or even impossible to reproduce results [151]. The need for a unified infrastructure has been addressed in several ways.

Most importantly, Gousios and Spinellis [63] designed and implemented a platform for integrated analysis of VCS, mailing list, and issue tracking data. The platform includes data collection and transformation into a common relational data model. Researchers can implement their own analyses as plug-ins which make use of the solid infrastructure able to parallelise tasks. However, the approach is processing centered and does not discuss the data extraction and transformation in detail.

Another unification approach is followed in [45] where a DSL and an infrastructure for MSR are presented. Using the proposed language *Boa*, programming efforts are greatly reduced and scalability and reproducibility of results are improved. However, the approach focuses only on VCS and does not discuss the issues related to the integration of various heterogeneous data sources. Data extraction from VCS alone is non-trivial [53].

Several works exist that aim to describe the data under study by means of ontologies. Facts are described as 3-tuples (subject, predicate, object). Keivanloo et al. [88] publish a large data collection of integrated VCS, issue tracking, and quality evaluation data on the web in Linked Data format. The work focuses on creating a common vocabulary and sharing knowledge, not on data extraction and transformation. Very close to our work is [92] where VCS, mailing list, and issue tracking data is represented in the *Web Ontology Language* (OWL). They use a similar meta-modelling approach and the structural part is also based on the *FAMOOS Information Exchange Model* (FAMIX) model. A layered extraction and transformation architecture that reuses existing tools that is similar to ours, is discussed in some detail in [61] and used in [60].

[39]See `http://repast.sourceforge.net`.

A mature platform for generic data analysis, visualization, and mining is *Moose*[40]. It is based on extensive meta-modelling, an extensible plug-in structure and a rich variety of existing tools. Although generic in principle, *Moose* focuses specifically on the analyses of software. At *Moose*'s core lies the FAMIX meta-model family that models object-oriented programs in a language independent manner. While FAMIX models focus on describing static snapshots of object-oriented systems, *Hismo* [56] enables the incorporation of historical information related to artifacts by means of model transformations. We follow a similar approach, relying on information obtained from FAMIX models, as well as meta-models derived from other information sources, and incorporating notions similar to those found in *Moose* and *Hismo*. However, we rely on facilities provided by EMF and related technologies to design and implement a flexible high-level infrastructure that can be integrated into *Eclipse*-based environments.

CODEMINE [32], a conceptually similar approach developed in parallel as a proprietary solution at Microsoft forgoes the assessment-specific abstraction, instead providing an API to the data store of extracted facts through a common data model, integrating all available facts. They also feature a set of platform services related to data cataloging, security and access permissions, event logging, data archiving, and data publishing. Apart from archiving and logging, such services have not yet been considered for the approach discussed in this chapter and may be the subject of future work.

Finally, Scheidgen [159] mentions modelling software repositories in EMF as an application of their EMF fragmentation technique. Our approach is similar in that we rely on a set of MSR meta-models developed on top of EMF, but while they focus primarily on methods for scaling large model instances, we focus on providing a flexible software mining infrastructure. The approaches can be considered complementary, in that our infrastructure will benefit from better handling of large model instances and at the same time, it can serve as a case study for different fragmentation strategies to improve the scalability of EMF models.

5.6. Summary

In this chapter, we discuss a high-level model-based approach to software mining. The approach is based on domain-specific meta-models related to assessment tasks of interest, describing the relevant concepts and their relationships as the common core information model. Facts needed for the assessment are extracted often by third-party approaches and tools where available, which results in heterogeneous facts assets. To ease the integration of these diverse heterogeneous facts assets, we translate them into homogeneous high-level facts model instances. These can then be assembled together and mapped to the high-level concepts in the assessment-specific meta-models by means

[40]See `http://www.moosetechnology.org`.

of model transformations, which are used to populate and enrich instances of the assessment meta-models in a stepwise manner. The assessment model instances are then queried to produce assessment assets which can be fed into third-party assessment applications for prediction, clustering, simulation, and visualisation purposes. The approach serves as a glue between different existing third-party and custom-made mining solutions, interconnecting the various tools and assets, related to both data extraction and knowledge derivation, at a high level of abstraction, without unnecessarily tight coupling. The proposed approach can provide traceability links between transformations in order to support validation of obtained results and actions upon these results, as well as extensibility at any point in the process. We presented a concrete instantiation for the realisation the approaches described in Chapter 4 and Chapter 3. We relied on various tools for facts extraction which are widely used in research. The modelling approach and the corresponding technologies provide interfaces for obtaining models from various lower-level representations by means of corresponding mappings which makes the integration of various input and output formats very convenient and reusable. The intention of the proposed approach is to lower the barrier to entry for researchers and practitioners alike and allow them to focus on the assessment tasks rather than the mining technicalities, without imposing any restrictions with regard to the available facts, their integration, and their application.

The presented instantiation showcase describes one concrete instantiation scenario related to approaches discussed in this thesis. The model-based software mining infrastructure can be tailored for other assessment tasks as well, even beyond the domain of software mining and software assessment. Other types of facts assets, such as test coverage reports, can be integrated in a similar manner to support the presented assessment task or related ones. Similarly, a different set of raw and facts assets can be considered for further assessment tasks, such as investigating the activity on mailing lists and ITS. In addition, other applications, can be integrated as well, reusing the facilities related to particular asset and facts types between instantiations.

Certain aspects remain the subject of further work. To support larger scale models and assessment tasks, viable and transparent scalability solutions need to be investigated. Deploying the mining infrastructure in a cloud environment is of particular interest for future work. We have started exploring a deployment of the approach within a cloud-based smart data platform for supporting empirical software research [174, 175]. While the approach supports a wide range of integration scenarios by extension of the flexible underlying transformation technologies, some assets and assessment scenarios may pose new integration challenges.

6. Case Studies

In this chapter, we describe case studies performed to evaluate the approaches for the identification of causes for events of interest described in Chapter 3 and for the characterisation of developer behaviour discussed in Chapter 4. In the following sections we focus on the description of the experiments and their results. In Chapter 7 we discuss the results and their interpretation in the context of this thesis.

6.1. Goals

The overall goal for the experiments is to demonstrate and evaluate the effectiveness of the approaches discussed in this thesis. We formulate specific goals aligned with the overall challenges and high-level questions identified within Chapter 1.

Before we address the main challenges, we need to assess the approach for the identification of potential causes for events of interest. The primary goal is to *asses the impact of the multi-layer approach*. For this goal, we investigate the potential benefits of using the multi-layer weighting approach at finer levels of granularity as compared to simply inheriting labels or even weights from containing artifacts across the used data sets. We evaluate the impact of different weight distribution strategies and thresholds. Based on the results, we select the strategies and thresholds that will be used for the experiments addressing the subsequent challenges.

In order to determine what constitutes developer behaviour, we considered the different circumstances under which developers operate, based on the various sources of information, levels of granularity, and collaborations with other developers. We defined different dimensions and characteristics related to the circumstances and grouped them under situational factors (related to the artifacts on which a developer works) and dispositional factors (related to the developer working on the artifacts). To *assess the impact of the different characteristics as well as the two groups of factors as a whole*, we focus on the impact of the different characteristics with respect to predictive modelling for defect prediction. We evaluate the additional characteristics across the different dimensions against a baseline of only considering part of the situational characteristics relating to the target state.

Next, we proceed to *assess the impact of the circumstances that are associated with potential causes for events of interest globally and for individual developers and/or groups of developers*. We focus on bug fixes as events of interest and corresponding bugs as their potential causes. We evaluate predictive modelling for defect prediction based on the activities of individual developers against a baseline considering the activities of all developers.

To investigate the impact of changes in the behaviour of developers, we identify similar activities in the behaviour of developers with respect to potential causes for bug fixes as events of interest. To achieve this, we need to *identify clusters of similar circumstances that are associated with potential causes for events of interest within the behaviour of individual developers and assess their impact with respect to potential causes for events of interest*. Correspondingly, we cluster activities based on their similarity and use them in predictive modelling for defect prediction against a baseline considering all activities.

Finally, to investigate transfer opportunities between different developers and different projects, we *assess the impact of using differentiated predictive models trained on the behaviour of one developer to predict causes for events of interest in the behaviour of other developers*. We use the activities of each developer to train a predictive model and predict the outcomes of activities of all other developers, both within the same project and within other projects. In some cases, the same developers contributed to multiple projects in the data sets, so we evaluated also using the activities of one developer within one project to predict the outcomes of the activities of the same developer within another project.

6.2. Evaluation Criteria

The case studies seek to assess the impact of the approaches discussed in this thesis with regard to several different goals. While the goals are related and rely on the same data, the different focus of the approaches requires different evaluation criteria. In this section we discuss the individual criteria for each approach.

6.2.1. Identifying Potential Causes for Events of Interest

To evaluate the impact of the multi-layer approach we consider several different aspects. As a baseline we first consider the scenario where no weighting is applied (binary classification of causes), that is every potential cause is considered as the cause for an event of interest. In this case, any revision at the project layer that is identified as the potential cause for an event of interest carries full responsibility. This applies also to all artifacts modified within that revision both at the file and at the logical layer (copy strategy).

Second, we consider the scenario where weighting is applied at the project layer based on the approach described in Section 3.3 (weighted classification of causes), that is potential cause is assigned a contributed weight for each event of interest it has contributed to. The contributed weight is divided among all potential causes. At the file and logical layers we still apply the copy approach for weight distribution.

Third, we consider the scenarios where both the weighting approach and the weight distribution strategies are applied at all layers based on the approach described in Section 3.5 (distributed classification of causes). In this case, the contributed weight is divided among all potential causes at the project layer, whereas the different weight distribution strategies are applied at the file and logical layers.

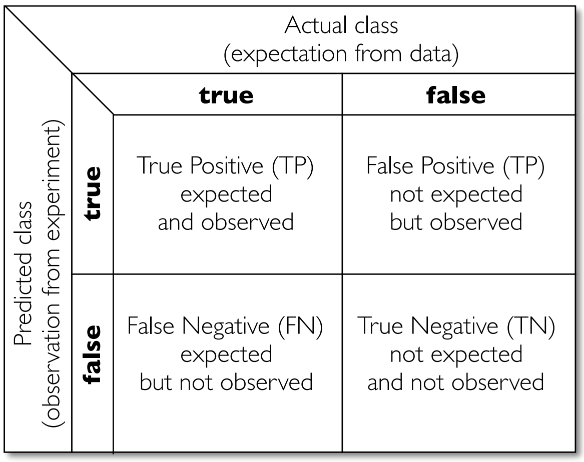
The main goal of the approach is to quantify and refine the identified causes for events of interest at different levels of granularity. We report and compare the distribution of the identified causes for each scenario and corresponding amount of artifacts and code that is potentially causing events of interest, focusing on the reduction of the number of artifacts and amount of code implicated in causing events of interest. We focus on one type of events of interest — bug fixes based on keyword search.

6.2.2. Developer-Centric Assessment

To evaluate the impact of the developer-centric assessment, we compare the outcomes of applying predictive modelling for defect prediction in different scenarios. Predictive models for defect prediction are typically evaluated based on common measures for the evaluation of binary classification approaches. Defect prediction can be considered as a binary classification problem, where a predictive model classifies data instances into defective (class *true*) or not (class *false*). The outcome from the classification is the observation from a defect prediction experiment. The actual classes for the data instances used for the evaluation are typically known in advance and define the expectation. The comparison between the observation and the expectation, or the predicted and actual classes after an experiment results in a so-called *confusion matrix*. The conceptual idea behind the confusion matrix is shown in Figure 6.1. In a binary classification experiment, there are four possible outcomes for each data instance:

- *true positive* (*tp*) if it observed and expected, that is both predicted to be defective and actually defective in the context of defect prediction,
- *false positive* (*fp*) if it observed but not expected, that is, it is predicted to be defective and but not actually defective (also referred to as *Type I error*),
- *false negative* (*fn*) if it is not observed, but expected, that is, it is predicted to be not defective, but it is actually defective (also referred to as *Type II error*,
- *true negative* (*tn*) if it not observed and not expected, that is, both predicted to be not defective and actually not defective in the context of defect prediction.

Based on the relationships between these outcomes, various measures are defined to assess the performance of a predictive model. Commonly used among them are *precision* (also known as *positive predictive value*) and *recall* (also known as *true positive*



Predicted class (observation from experiment)		Actual class (expectation from data)	
		true	**false**
	true	True Positive (TP) expected and observed	False Positive (TP) not expected but observed
	false	False Negative (FN) expected but not observed	True Negative (TN) not expected and not observed

Figure 6.1.: Confusion matrix: expectations and observations in binary classification

rate and *probability of detection*). Precision in this context is indicative of the proportion of outcomes predicted as defective that are actually defective and defined as:

$$precision = \frac{tp}{tp + fp} \tag{6.2.1}$$

Recall in this context is indicative of the proportion of actual defective outcomes that are predicted as such and defined as:

$$recall = \frac{tp}{tp + fn} \tag{6.2.2}$$

Considering *precision* and *recall* as indicative of the performance of a predictive model with respect to *false positives* and *false negatives* is more transparent and easily interpretable. However, contemplating both measures over extensive experiments may be too cumbersome. The *F-measure* (also known as *F-score*) as the harmonic mean between the *precision* and *recall* is well suited for summarising and comparing results from multiple experiments in a more concise way. It is defined as:

$$F - measure = 2 \cdot \frac{precision \cdot recall}{precision + recall} \tag{6.2.3}$$

These measures are frequently used in the evaluation of defect prediction approaches, however, there are other measures that are also commonly used, including *area under* *the curve* (both based on *receiver operating characteristic* and *precision-recall* curves), *G-measure*, *Matthews' Correlation Coefficient*, among others. Currently, there is no common agreement regarding the best suited measures for the evaluation of predictive models and in particular when applied to defect prediction [75, 85]. Weighted averages of *precision*, *recall*, and *F-measure* can be considered as indicative of the discrimination ability of a predictive model. However, due to the typically low prevalence of activities resulting in a defect in the data sets used for defect prediction, weighted averages can mask poor performance with respect to correctly predicting activities resulting in a defect. Correspondingly, we focus on the *precision*, *recall*, and *F-measure* for the *true* class only, that is for activities resulting in a defect. Nonetheless, we collected various other performance measures during the experiments which will be the subject for further analysis in the future.

6.3. Data Sets Description

The projects used for the case studies were selected on randomised basis from the *K Desktop Environment* (KDE), Apache, and Eclipse communities. These communities were selected as some of the largest sources of mostly homogeneous data due to established guidelines for contributors and supporting infrastructure. The main criteria for the selection of projects were the use of the *git* VCS, implementation in the C++ and Java languages, as well as the use of the respective BugZilla ITS for each community (indicated by the presence of a corresponding BugZilla project with at least 10 reported issues). The projects were selected in such a way that they represent a mixture of different classes with respect to number of revisions, number of developers, and number of reported issues.

The VCS and ITS assets related to the projects were processed in order to extract basic facts and transform them into facts models according to the model-based mining approach described in Chapter 5. This resulted in a separate data set for each project comprising an *MG* model from the VCS, a *BZ* model from the ITS, as well as a set of *FAMIX* models containing static analysis data for each revision in the VCS. An overview of the resulting data sets is presented in Table 6.1. The selected projects are grouped by language (C++ and Java) and sorted by number of revisions in each group. The time span indicates the years of the first and last recorded revision in the data set. Since most projects are still in active development, collecting the data at a later point will likely result in different numbers. The number of developers indicates the size of the contributor community which is typically roughly proportional to the number of revisions, however, there are also some anomalies, such as *egit* and *jgit*, which have unusually high number of developers for the corresponding number of revisions. This suggests that a lot of the contributors participated in the project only sporadically. Finally, the number of issues is an indication of the community involvement in requesting

Project	**Revisions**	**Time Span**	**Developers**	**Issues**	**Lang.**
amarok	32823	2003–2012	349	17181	C++
kate	15112	2001–2014	352	7263	C++
konsole	6426	1998–2014	262	4153	C++
k3b	6217	2001–2012	129	3832	C++
ktorrent	4129	2005–2014	77	2576	C++
rekonq	2814	2008–2012	78	2140	C++
plasma-nm	1662	2013–2014	36	165	C++
ksudoku	802	2007–2014	69	93	C++
yakuake	516	2006–2014	28	267	C++
ant	15321	2000–2014	50	5849	Java
emf	8690	2004–2014	23	2574	Java
poi	6125	2002–2014	34	3140	Java
log4j	3551	2000–2014	21	1365	Java
egit	3219	2009–2014	84	2803	Java
jgit	2558	2010–2014	111	669	Java
sirius	701	2013–2014	16	132	Java
egit-github	592	2011–2014	15	2803	Java

Table 6.1.: Overview of selected data sets.

new features and reporting issues related to the project. While a lot of the issues are typically reported by the developers themselves for the purposes of project management and transparency, depending on the type of the project and its target audience, the number of issues can reflect the involvement and size of the broader community, including developers of other projects relying on a particular project and end users. For example, most of the C++ projects selected from the KDE community are targeted towards end users, whereas most of the Java projects selected from the Apache and Eclipse communities are targeted towards other developers which integrate them into their own projects. The scope, purpose, and distinctive characteristics of the selected projects are summarised below:

amarok Amarok (`amarok.kde.org`) is a comprehensive music and media playback and management software. It is integrated in the KDE package and as such bundled with many Linux distributions hence exposed to a large number of end users. Its first release was published in 2003.

kate Kate (`kate-editor.org`) is an advanced text editor delivered as part of the KDE package. Similar to Amarok, it is bundled with many Linux distribution and hence also exposed to a large number of end users. In addition, it is possible to embed Kate as an editing component in other KDE applications, thus Kate serves both end users and downstream developers. Its first release was published in 2001.

konsole Konsole (`konsole.kde.org`) is a terminal emulator integrated in the KDE environment. Due to its nature it is targeted towards more technical end users. Similar to Kate, it can be integrated in other KDE applications in order to provide embedded terminal functionality, thus it also serves downstream developers within the KDE community. First recorded tagged revision in the VCS repository referring to v1.1.0 is from 1999.

k3b K3b (`k3b.org`) is a CD and DVD authoring application bundled with the KDE environment. While it has been around reportedly since 1998, the first VCS records are from 2001.

ktorrent Ktorrent (`ktorrent.org`) is a BitTorrent client delivered with the KDE environment. Version v1.0 was tagged in the VCS repository in 2005.

rekonq Rekonq (`rekonq.kde.org`) is a lightweight web browser developed within the KDE environment. While the larger KDE projects are generally bundled with KDE and serve as default applications for the corresponding purposes, rekonq an optional application that may be installed in addition to the default web browser bundled with KDE. Given the wider popularity of other cross-platform browsers such as Firefox and Chrome, and the presence of another browser included by default with KDE, rekonq has remained comparatively small and as of 2014 it is no longer actively developed. Its initial release was published in 2008.

plasma-nm The Plasma Network Manager applet is a small project providing a GUI front-end for managing network connections within the KDE environment. It was first released in 2013.

ksudoku Ksudoku (`games.kde.org/game.php?game=ksudoku`) is a logic-based symbol placement puzzle game for the KDE environment. The first recorded revisions in the VCS repository are from 2007.

yakuake Yakuake (`yakuake.kde.org`) is a terminal emulator for the KDE environment. Similar to rekonq, it is an optional application providing a different set of features and as such it is exposed to a smaller audience of end users. However, similar to Kate and Konsole it may be integrated into other applications, thus also serving downstream developers within the KDE community. The first recorded revisions in the VCS repository are from 2006.

ant Apache Ant (`ant.apache.org`) is a tool for automating software build processes. Its end users are typically software developers, although it may be integrated in products targeting a wider audience. It saw its first release in 2000.

emf EMF (`eclipse.org/emf`) is an Eclipse-based modelling platform providing facilities for the specification of structured data models, as well as instantiating and manipulating model instances. It provides a foundation for the interoperability between EMF-based tools and applications. While its direct users are software developers, it is also often integrated into various end users software products. The first recorded revisions in the VCS repository are from 2004.

poi Apache Poi (`poi.apache.org`) is a collection of libraries providing Java interfaces for reading, writing, and manipulating Microsoft Office documents. Its direct users are software developers, but it is usually integrated into end user software products. While its first version was released in 2001, the VCS repository provides records going only as far back as 2002.

log4j Apache Log4j (`logging.apache.org/log4j`) is a Java-based logging framework. Similar to Apache Poi, while it is targeted at developers as its direct users, it is often integrated in a wide range of end user software products. Its first version was released in 2001.

egit EGit (`eclipse.org/egit`) is a plug-in for the Eclipse platform providing an integration with the Git VCS. It relies on JGit as a back-end providing an interface to the VCS. Its primary users are developers working with the Git VCS within Eclipse. The first recorded revisions in the VCS repository are from 2009.

jgit JGit (`eclipse.org/jgit`) is a lightweight Java-library for working with Git VCS repositories. Similar most of the Java projects, it is targeted at developers as its direct users, and even users of downstream applications using JGit are often developers. The first recorded revisions in the VCS repository are from 2010.

sirius Sirius (`eclipse.org/sirius`) is a platform for creating custom graphical modelling workbenches built on top of other Eclipse modelling technologies, such as EMF and the *Graphical Modeling Framework* (GMF). Similar to other selected Java projects, it is targeted at developers as its direct users, but it is also integrated in a wide range of end user software products. The first recorded revisions in the VCS repository are from 2013.

egit-github EGit-Github (`eclipse.org/jgit`) is an extension for EGit providing additional integration for working with Git VCS repositories hosted on Github. The first recorded revisions in the VCS repository are from 2011.

6.4. Case Study Results

In this section we describe the results from two case studies performed to evaluate different aspects of the approaches for the identification of causes for events of interest described in Chapter 3 and the characterisation of developer behaviour discussed in Chapter 4.

6.4.1. Identifying Potential Causes for Events of Interest

In this section, we investigate the impact of using the multi-layer weighting approach at finer levels of granularity and of the different weight distribution strategies. First we investigate the potential benefits of using the layered approach at finer levels of granularity in comparison to inheriting weights from containing artifacts. Weights are inherited to all contained artifacts under the assumption that without any refinement all states of contained artifacts at the file and logical layer corresponding to a given state at the project layer identified as a potential cause for an event of interest. This results in a set of states projected to be contributing to causing an event of interest. In contrast, the layered approach distributes weights on each layer based on independent cause-fix relationships.

6.4.1.1. Methodology

To assess the impact of the layering, we initially consider 2-fold grouping of states of artifacts, where states are grouped based on their average weights as either having a weight of 0 (not considered as a cause for an event of interest) or 1 (including all states considered as causes for events of interest where their weight is more than 0). We compare the number of states identified as causes for events of interest when using the layered approach at finer levels of granularity and the baseline approach of inheriting weights from containing artifacts. This way, we can determine the number of states that would otherwise be incorrectly identified as causing events of interest resulting in noise in the data if the baseline approach is used.

To gain further insight into the impact of the weighting approach we consider 10-fold grouping. In this case, we assign states considered as causing events of interest to 10 groups based on their weight. The groups represent 10 intervals between 0.0 and 1.0, where each interval has a lower (*lb*) and an upper (*ub*) bound, for example the for the interval between 0.0 and 0.1, $lb = 0.0$ and $ub = 0.1$. The states are assigned to the corresponding intervals based on $lb < aw \leq ub$. This provides us with a summarised insight into the distribution of the weights among the states. It also can help in prioritising high confidence causes that have higher average weight and filtering out low confidence causes that can also be considered noise.

```
"defect(s)?"
"patch(ing|es|ed)?"
"bug(s|fix(es)?)?"
"(re)?fix(es|ed|ing|age|\s?up(s)?)?"
"debug(ged)?"
"\#\d+"
"back\s?out"
"revert(ing|ed)?"
```

Listing 6.1: Regular expressions for the BugFix factor

Finally, we assess the impact of the weight distribution strategies. Since the strategies do not affect the binary grouping, we consider their impact based on the 10-fold grouping. We compare the differences in the distributions of average weights from the projection approach when using the different weight distribution strategies.

6.4.1.2. Layered Approach

We consider the *BugFix* factor. It is based on the *is_bug_fix* property of the *Revision* in the *MG* model. *Revisions* in the *MG* model are mapped to *GlobalStates* in the *CFA* model and correspond to states at the project layer. The *BugFix* factor has the value 1 if the *message* of the *Revision* matches one of the regular expressions defined in Listing 6.1 and 0 otherwise. The regular expressions are adopted from the *CVSAnalY/MininGit* tool that was used for the extraction of facts from the VCS.

As an example, we first contemplate a 2-fold grouping scenario to showcase the differences between the layered approach and a projection from the project layer to finer levels of granularity. Consider the summary for the *BugFix* factor for the randomly selected *Konsole* project shown in Table 6.2. At the project level of granularity we consider only the project states that involve the modification of artifacts of type *code* at the file level of granularity. Naturally, in this case there is no difference between the layered approach and the projected number of states since the layering approach does not affect the project layer. At the project layer we observe that 1509 states or 30.7% of the states are considered to be causes for events of interest with regard to the *BugFix* factor (i.e. considered to be causes for bug fixes, or, put simply, considered to be adding bugs that needed fixing). If we do not apply the layering approach, this would implicate 5485 states or 46.5% of the states at the file layer as projected to be causing bug fixes. As noted in Section 3.4, it is often the case that only a subset of the artifacts at the file level of granularity contribute to a cause for an event of interest for a given state at the project level of granularity. Applying the layered approach results in 2616 states or 22.2% of the states at the file layer of granularity. Thus, in this case the layered approach helps to reduce the number of states incorrectly considered to be causing bug fixes by 52.3%, which would otherwise introduce noise at the file level of granularity.

		Projected		Layered		
Group	**Total**	**Count**	**Ratio**	**Count**	**Ratio**	**Change**
Project (code)	4915	1509	30.7%	1509	30.7%	0.0%
File (code)	11794	5485	46.5%	2616	22.2%	-52.3%
Logical (Class)	2757	1999	72.5%	432	15.7%	-78.4%
Logical (Method)	15584	10999	70.6%	1865	12.0%	-83.0%

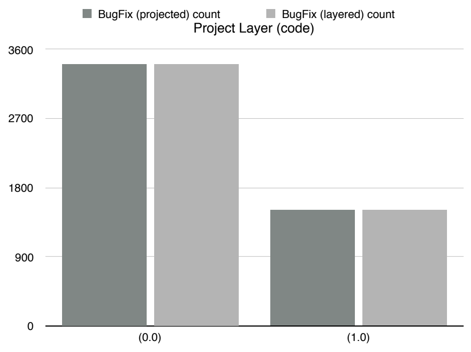
Table 6.2.: Causes for events of interest: Konsole (2-fold, factor BugFix).

The differences at the logical level of granularity are even more substantial with reductions of 78.4% and 83.0% in the number of states considered as causes for bug fixes for artifacts of type *Class* and *Method*, respectively. Considering the fact that without the layered approach more than 70% of the states would have been projected to be causing bug fixes, where in fact only 12–15% should be considered as such, as the remaining have no direct relationships with the bug fixes. In this case the layered approach makes a clear difference in reducing noise. These findings for the *Konsole* project are also visualised in Figure 6.2 where we see comparisons of the respective number of states causing events of interest with regard to the *BugFix* factor in the different layers when using projection from the project and when using the layered approach.

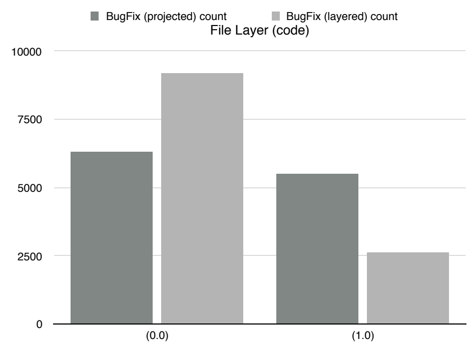
To emphasise the impact of the layered approach on the binary grouping, the amount of changes in the grouping of the states for the *Konsole* project at the file and logical layer are also visualised in Figure 6.3. In this case we see the percentage of states that were reassigned in each group when using the layered approach in comparison to the projected approach.

A summary of the results in the 2-fold grouping over all projects is shown in Table 6.3 (file layer) and Table 6.4 (logical layer)[41]. We focus on the grouping in the interval between 0.0 and 1.0 as it is of primary interest. The grouping for the states that are not considered as causes for events of interest can be inferred based on the provided counts and ratios. At the file layer, the application of the layered approach leads to reduction in the number of potential causes for events of interest in the range between 50% and 80%, with the *Egit-github* project reaching even 93%. This way the ratios of causing states are reduced from 45–80% to 10–30%, considerably reducing the amount of noise. At the logical layer, the reduction in the number of potential causes for events of interest is even more substantial, ranging from 75% to 95%. This leads also to corresponding reduction in the ratios of causing states from 50–95% to 4–18%, thereby also reducing the amount of noise. The *Egit-github* project again presents some anomalous results,

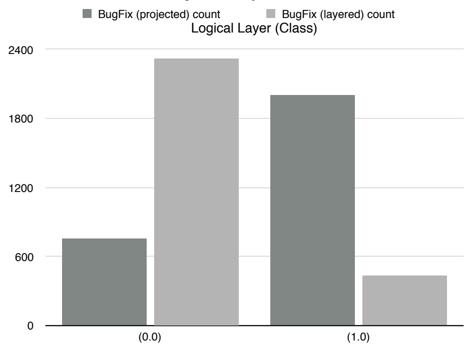
[41] Logical layer results for the projects `amarok`, `ant`, `emf`, `kate`, and `poi` were not available at the time of writing. Based on the results for these projects at the file layer as well as the overall trend for the other projects at the logical layer, the results for these projects at the logical layer are expected to follow the overall trends



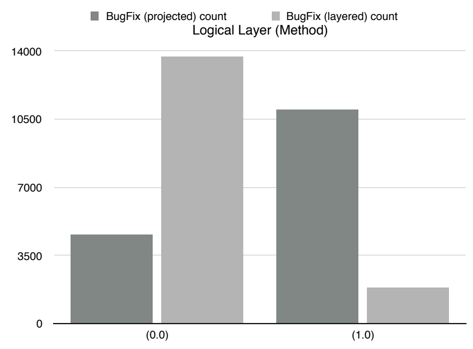
(a) Project layer (code)



(b) File layer (code)

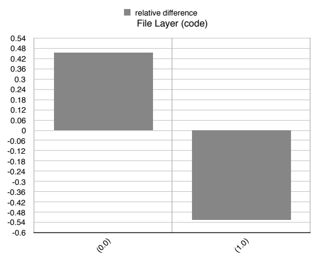


(c) Logical layer (Class)

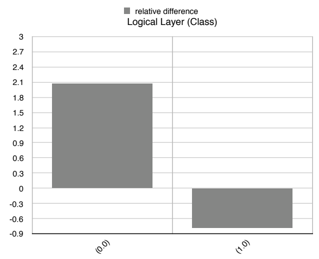


(d) Logical layer (Method)

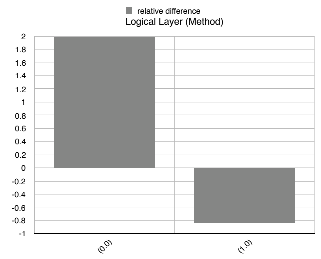
Figure 6.2.: Causes for events of interest: Konsole (2-fold, BugFix)



(a) File layer (code)



(b) Logical layer (Class)



(c) Logical layer (Method)

Figure 6.3.: Changes to causes for events of interest: Konsole (2-fold, BugFix)

	Projected		Layered		
Project	**Count**	**Ratio**	**Count**	**Ratio**	**Change**
amarok	52622	0.635	22188	0.268	-57.8%
kate	15201	0.466	6132	0.188	-59.7%
konsole	5485	0.465	2616	0.222	-52.3%
k3b	22420	0.718	9056	0.290	-59.6%
ktorrent	11668	0.552	2681	0.127	-77.0%
rekonq	5230	0.697	2464	0.329	-52.9%
plasma-nm	2015	0.488	478	0.116	-76.3%
ksudoku	653	0.587	267	0.240	-59.1%
yakuake	317	0.437	151	0.208	-52.4%
ant	26246	0.404	4798	0.074	-81.7%
emf	12194	0.463	2539	0.096	-79.2%
poi	25854	0.677	9997	0.262	-61.3%
log4j	8819	0.580	3359	0.221	-61.9%
egit	4808	0.584	1583	0.192	-67.1%
jgit	4094	0.478	990	0.116	-75.8%
sirius	19300	0.796	6088	0.251	-68.5%
egit-github	177	0.112	12	0.008	-93.2%

Table 6.3.: Overview of results at the file layer (code, 2-fold, BugFix).

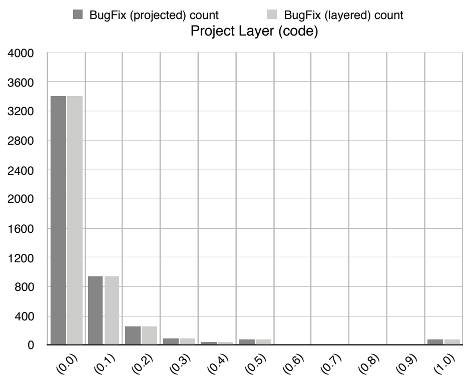
where the ratios of states causing events of interest to states not causing events of interest are much lower than the rest of the projects. Nonetheless the reduction resulting from the application of the layered approach falls in the range of the other projects.

6.4.1.3. Weighting Approach

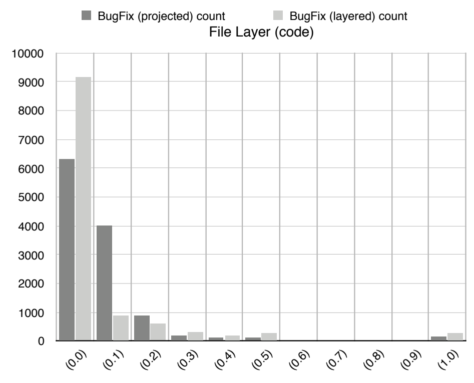
Next, we contemplate a 10-fold grouping scenario to showcase the weight distribution differences between the layered approach and the projection from the project layer to finer levels of granularity. Consider the summary for the *BugFix* factor for the *Konsole* project shown in Figure 6.4. As in the 2-fold grouping, the weight distribution has no impact at the project layer, however, there are noticeable differences at the file and logical layers. Notably, at the lower end of the spectrum, with the layered approach there is a sharp reduction in the number of states that are assigned an average weight in the interval between 0.0 and 0.1. At the same time there is a visible increase in the number of states that are assigned an average weight in the interval between 0.9 and 1.0 when using the layered approach. Consequently, this results in a lower number of low confidence causes and at the same time in a higher number of high confidence causes, both of which are desirable outcomes of the approach.

	Projected		Layered		
Project	**Count**	**Ratio**	**Count**	**Ratio**	**Change**
Class					
konsole	1999	0.725	432	0.157	-78.4%
k3b	9256	0.899	1344	0.130	-85.5%
ktorrent	5057	0.642	354	0.045	-93.0%
rekonq	1339	0.826	244	0.150	-81.8%
plasma-nm	466	0.612	29	0.038	-93.8%
ksudoku	347	0.792	58	0.132	-83.3%
yakuake	127	0.641	31	0.157	-75.6%
log4j	6442	0.619	1237	0.119	-80.8%
egit	6563	0.597	1709	0.155	-74.0%
jgit	4482	0.510	1014	0.115	-77.4%
sirius	13294	0.867	2682	0.175	-79.8%
egit-github	168	0.100	11	0.007	-93.5%
Method					
konsole	10999	0.706	1865	0.120	-83.0%
k3b	42523	0.887	5406	0.113	-87.3%
ktorrent	27940	0.610	1719	0.038	-93.8%
rekonq	7011	0.800	1335	0.152	-81.0%
plasma-nm	2121	0.619	247	0.072	-88.4%
ksudoku	1794	0.827	195	0.090	-89.1%
yakuake	1123	0.734	172	0.112	-84.7%
log4j	17146	0.593	1404	0.049	-91.8%
egit	13776	0.681	1943	0.096	-85.9%
jgit	17561	0.639	1268	0.046	-92.8%
sirius	64926	0.947	3932	0.057	-93.9%
egit-github	523	0.104	18	0.004	-96.6%

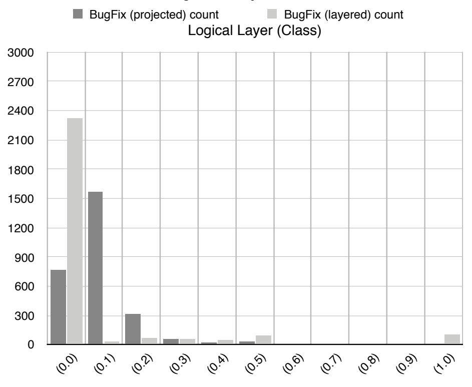
Table 6.4.: Overview of results at the logical layer (2-fold, BugFix).



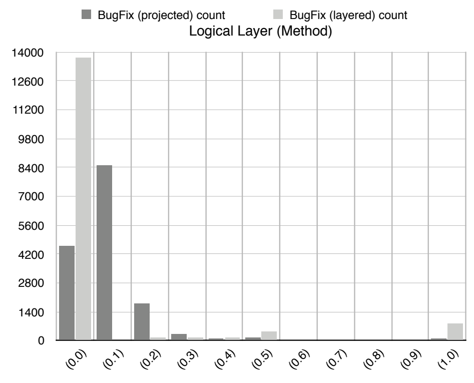
(a) Project layer (code)



(b) File layer (code)



(c) Logical layer (Class)



(d) Logical layer (Method)

Figure 6.4.: Causes for events of interest: Konsole (10-fold, BugFix)

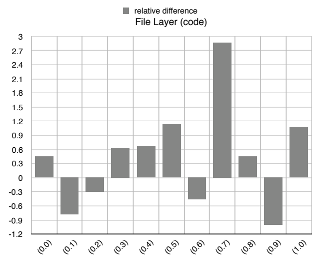
To have a more detailed look at the changes in each group, we consider the amount of changes in the respective groups visualised in Figure 6.5. In the provided example, we can observe increases in the number of states corresponding to the increase in confidence. At the file layer, there are two exceptions (0.5 to 0.6 and 0.8 to 0.9) where there is a decrease in the number of causes with corresponding confidence. At the logical layer for artifacts of type *Class*, the groups for the intervals between 0.3 and 0.4 as well as 0.5 and 0.6 showed little or no difference, and in the groups for the intervals between 0.6 and 0.7 as well as 0.8 and 0.9, the projection approach did not produce any causes with that confidence, whereas the layered approach did produce 11 causes leading to an increase. For the type *Method*, there is only one decrease for the interval between 0.5 and 0.6.

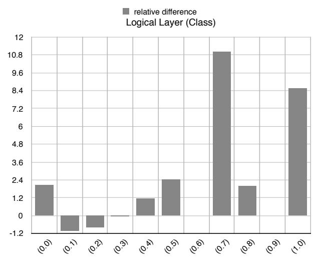
Examining the results related to the amount of change in the 10-fold grouping for two other randomly selected examples for projects *Ktorrent* and *Log4j*, shown on Figure 6.6 and Figure 6.7, respectively, we obtain a somewhat similar impression. When using the layered approach, the number of states in the high confidence groups for the interval between 0.9 and 1.0 has increased substantially at the logical layer for artifacts of the *Class* type for both projects, as well as for artifacts of the *Method* type in the *Ktorrent* project, whereas there is only a marginal increase for artifacts of the *Method* type in the *Log4j* project. At the same time there is an overall decrease in lower confidence groups. However, there are also some notable differences. At the logical layer, for the *Log4j* project there is also a substantial increase in the medium confidence groups for the interval between 0.4 and 0.5 for artifacts of both types. At the file layer for the project *Ktorrent* there is a slight decrease (-24.4%) in the high confidence group for the interval between 0.9 and 1.0, which is rather atypical when compared to all other projects. This suggests that among the states projected to be causing events of interest with high confidence at the file layer, there is still a considerable amount of noise. This is due to the fact that even states causing events of interest with high confidence the project layer frequently contain multiple states at the file layer, not all of which are related to causing the event of interest. Ultimately, while the layered approach generally leads to redistribution of weights towards increasing the number of states considered as causing events of interest with high and medium confidence and decreasing the number of states considered as causing events of interest with low confidence, the distribution of the weights may still vary between projects depending on the nature of changes developers commit.

A summary over all projects of the 10-fold grouping for the changes in the number of causes for events of interest when comparing the layered approach is shown in Figure 6.8. The overall trends are largely similar to the individual projects discussed above. We note that smaller projects tend to exhibit larger changes compared to larger and more mature projects.

6.4.1.4. Weight Distribution Approach

So far we considered the case where no weight distribution strategy is applied, that is, the removed weights for events of interest at the project layer are inherited by the corresponding states at the file and logical layers. Next, we contemplate the impact of the different strategies on the distribution of causes for events of interest. Similar to the results in Figures 6.5, 6.6, and 6.7 discussed above, we consider the changes to the number of causes for events of interest when compared to the projection approach. We compare the differences in distribution of the causes for events of interest within the 10-fold grouping from the projection from the project layer to finer levels of granularity. We consider the scenario when using the layered approach without any weight distribution strategy (inheriting the removed weight) as discussed above, and when using the layered approach with the different weight distribution strategies described in Section 3.5.


(a) File layer (code)


(b) Logical layer (Class)

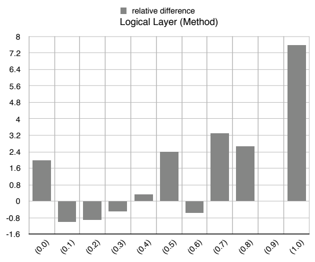
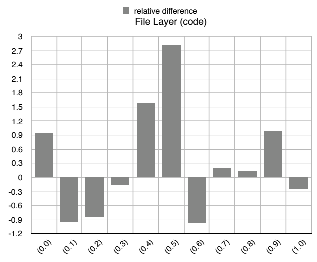
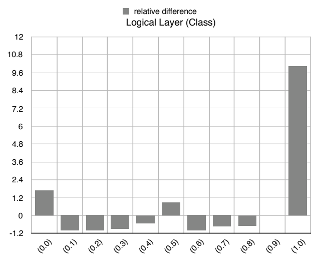

(c) Logical layer (Method)

Figure 6.5.: Changes to causes for events of interest: Konsole (10-fold, BugFix)


(a) File layer (code)


(b) Logical layer (Class)

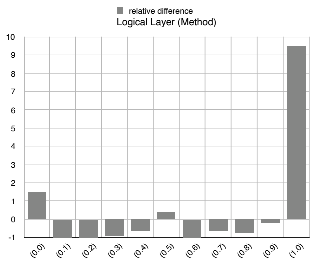
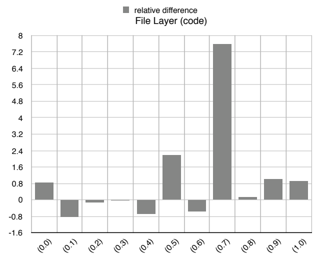
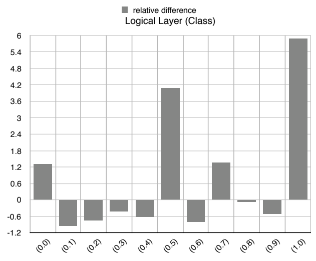

(c) Logical layer (Method)

Figure 6.6.: Changes to causes for events of interest: Ktorrent (10-fold, BugFix)


(a) File layer (code)


(b) Logical layer (Class)

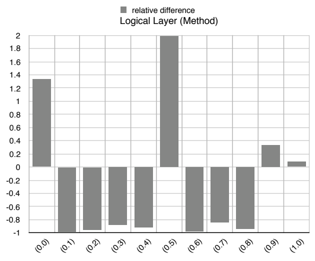
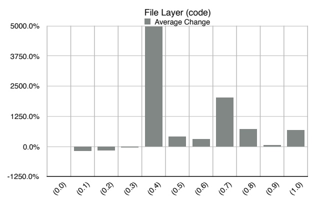
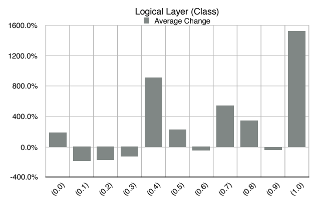

(c) Logical layer (Method)

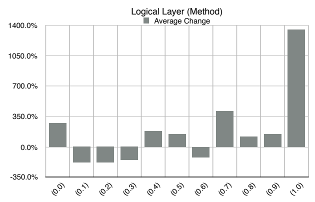
Figure 6.7.: Changes to causes for events of interest: Log4j (10-fold, BugFix)



(a) File layer (code)



(b) Logical layer (Class)



(c) Logical layer (Method)

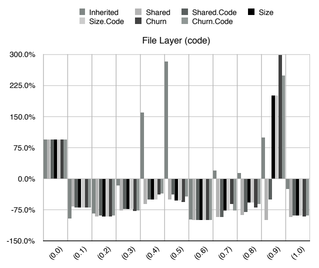
Figure 6.8.: Average changes to causes for events of interest (10-fold, BugFix)

The results for the three projects discussed above are summarised in Figures 6.9, 6.10, and 6.11. Several observations can be made based on the three projects. The use of weight distribution strategies reduces the number of high confidence causes (in the interval between 0.9 and 1.0) when compared to not using weight distribution strategies (and in some cases even when compared to the projection approach). This is expected, as where removed weights in the fixing states are inherited from the project state, a weight distribution strategy splits the removed weight among all states at a finer level of granularity, and in most cases changes affect multiple artifacts. All strategies have largely similar effect on lower confidence causes (in the groups between 0.1 and 0.5), however, the effects on higher confidence causes (in the groups between 0.6 and 0.9) vary between projects and levels of granularity. This is a desirable outcome as different strategies are intended to highlight different aspects of fixes for causes of events of interest in order to facilitate prioritisation based on a particular characteristic. Depending on the nature of the changes in a project, these become evident also in the summarised results.

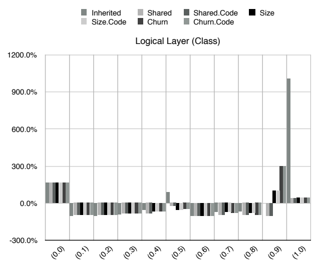
A more detailed view on the distribution of the average weights across the different layers for the *Konsole* project is shown in Figures 6.12, 6.13, and 6.14. In this case, each state of each artifact at the corresponding layer is plotted on the horizontal axis and the average weights are plotted on the vertical axis. Different colours and symbols are used to distinguish the average weights when using the different distribution strategies. Black dots are used to indicate the average weights when using inherited weights (no weight distribution). From the detailed view, we can observe that without weight distribution, the resulting average weights tend to form narrow bands at 1.0, 0.5, 0.33, and 0.25 (especially at the file layer and at the logical layer for type *Method*). On the other hand, the use of weight distribution strategies provides a more nuanced view with weights spread across a wider spectrum.

6.4.1.5. Other Factors

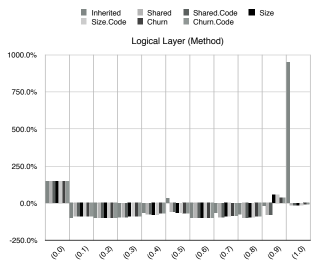
So far we focused only on the *BugFix* factor discussed above. In addition to it, we explored several other factors. They were based on simpler regular expressions



(a) File layer (code)

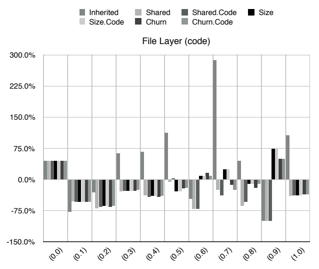


(b) Logical layer (Class)

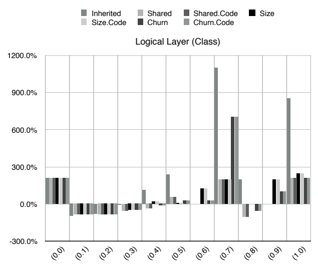


(c) Logical layer (Method)

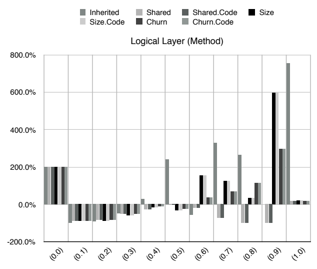
Figure 6.9.: Changes to causes for events of interest: Ktorrent (10-fold, BugFix, with distribution strategies)



(a) File layer (code)

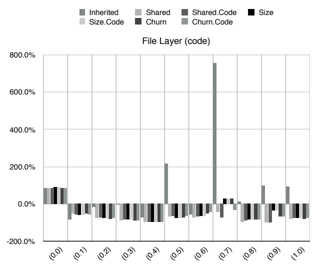


(b) Logical layer (Class)

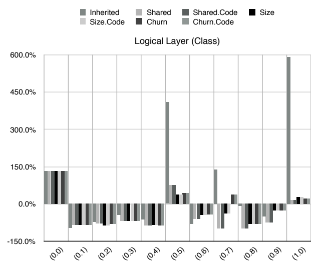


(c) Logical layer (Method)

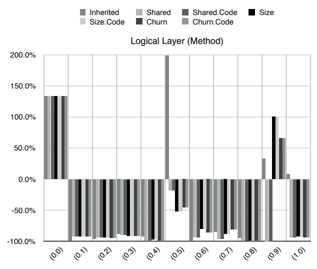
Figure 6.10.: Changes to causes for events of interest: Konsole (10-fold, BugFix, with distribution strategies)



(a) File layer (code)



(b) Logical layer (Class)



(c) Logical layer (Method)

Figure 6.11.: Changes to causes for events of interest: Log4j (10-fold, BugFix, with distribution strategies)

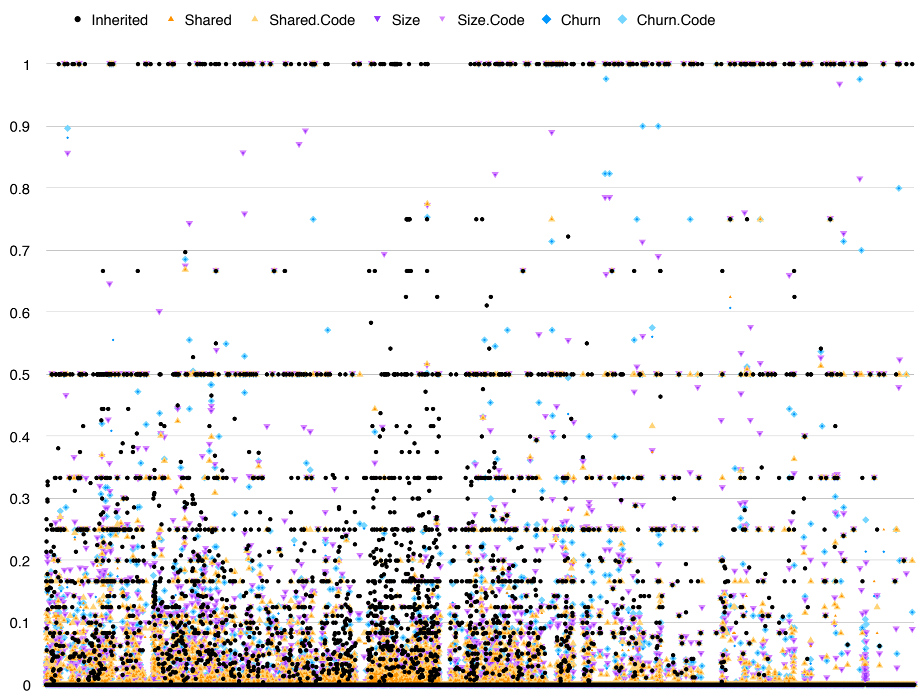


Figure 6.12.: Average weights for causes for events of interest with distribution strategies: Konsole (File layer (Code))

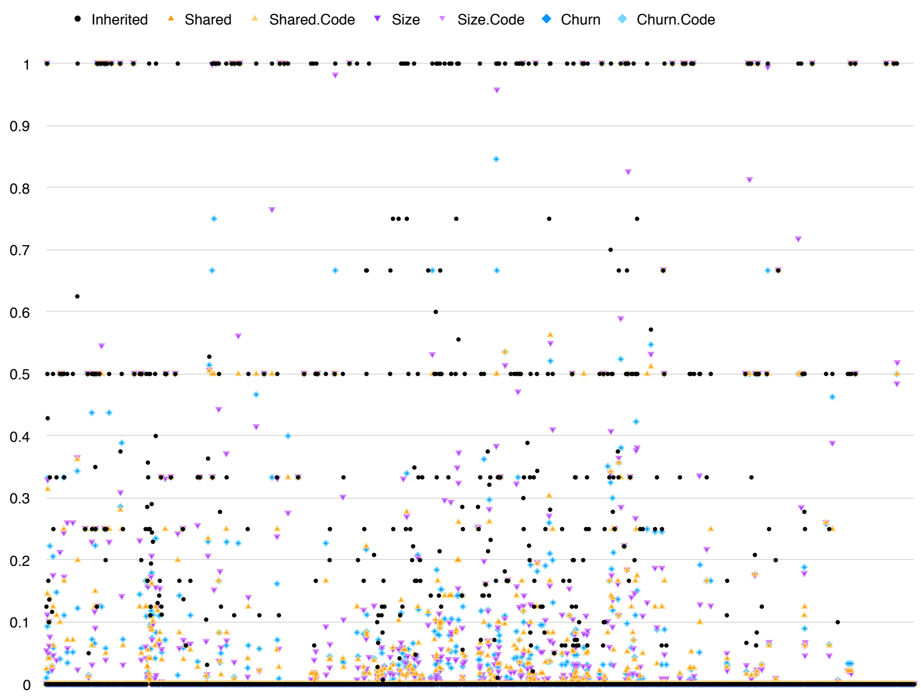


Figure 6.13.: Average weights for causes for events of interest with distribution strategies: Konsole (Logical layer (Class))

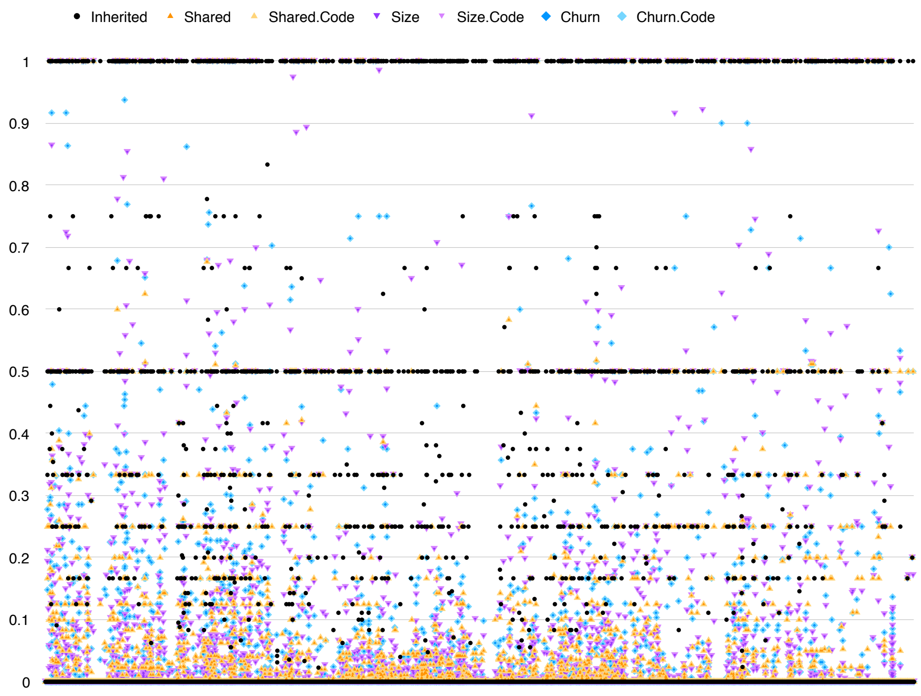


Figure 6.14.: Average weights for causes for events of interest with distribution strategies: Konsole (Logical layer (Method))

such as the *Fix* factor matching commit messages against the regular expression “`.*(fix|bug|bug:).*`”. The *Fix* factor yielded comparable results as the more elaborated *BugFix* factor, with only occasional minor deviations. This prompted the investigation of the *Refactoring* factor based on a similarly simple regular expression “`.+(factored|factoring).*`” for detecting potential refactorings. However, this approach yielded very few results across all projects (typically $< 5\%$ at the project layer and $< 1\%$ at the logical and file layers). Further investigation into refactorings needs to be considered in order to determine whether refactorings are in fact very rare in the contemplated projects or whether a more effective classification approach, such as the ones described in [14, 24, 68, 79, 183], needs to be applied to improve the results. The weighting approach itself is independent from the assignment of the removed weights for events of interest across the different factors. It only serves for the identification of the potential causes for these events of interest.

6.4.2. Developer-Centric Assessment

In this section we describe the results from the case studies performed to evaluate different aspects of the developer-centric assessment based on the characterisation of developer behaviour discussed in Chapter 4. While we aim to cover several aspects of the approach, the experiments discussed in this section and the extent to which the results are presented are still limited in scope and not exhaustive. We have reduced the results to F-measure averages comparisons for summarisation due to the broad scope of the evaluations. Considering other measures as well as looking beyond the averages merit additional discussion which is necessary to better understand the consequences of the different options described in this chapter.

For the evaluation, we considered data from all projects except *egit-github*, which was left with only three causes for events of interest after excluding activities from the confidence window. The ratio of causes for bug fix events of interest to total number of activities ranged between 6% and 33% (code) at the file layer and between 7% and 19% (class) as well as 3% and 19% (method) at the logical layer.

For each experiment, we split the activities into training and testing data. We followed a percentage split (*PS*) approach, splitting the data at 50%. In general, the activities included a low number of causes for events of interest resulting in a strong bias in the data. In addition, the causes for the events of interest are not evenly distributed throughout the recorded timeframes. Thus, in addition to the plain percentage split, we also consider *true split* (*TS*), where the activities are split based on the percentage of the causes for events of interest. In this case, the split is performed after reaching 50% of the causes for events of interest. This resulted in splits at points between 11% and 52% of the activities when considering all the activities in a project, and between 9% and 92% of the activities when considering the activities for each individual developer.

We considered five algorithms which are frequently used in the literature and are representative of different classes of machine learning approaches: C4.5 decision tree (*DT*) [143], logistic regression (*LR*) [31], naïve Bayes (*NB*) [156], random forest (*RF*) [17], and support vector machine (*SVM*) [178]. During the experiments, we used the default parameters for all of the machine learning algorithms as specified in the *Weka* [64] platform.

We applied the machine learning approaches on all activities within the project (referred to as *All*), as well as on the subsets of activities for each developer (referred to as *DS*). We considered only developers that have performed at least 100 activities and at least one activity caused an event of interest. At the file level of granularity, such developers performed 93% of the activities and 94% of the causes for events of interest across all projects, ranging between 76% and 99% for the individual projects. We summarised the results for the individual developers by taking the averages among all developers. Additionally, the low overall number of causes for events of interest translates to even smaller number of causes for events of interest for individual developers with fewer activities and thus resulting in a stronger bias in the data. Consequently, we also report the average of developers having at least 100 causes for events of interest in the training data (referred to as *DS 100*). At the file level of granularity, such developers performed 70% of the activities and 76% of the causes for events of interest across all

	Activities		**Causes**		
Application	**Count**	**% of All**	**Count**	**% of All**	**Causes Ratio**
All	263582	100%	49196	100%	18.7%
DS	245375	93.1%	46376	94.3%	18.9%
DS 100	186603	70.8%	37406	76.0%	20.0%

Table 6.5.: Overview of the data at the file layer (code)

projects, ranging between 22% and 95% for the individual projects. The causes for bug fixes amounted to 18% of all activities in the *All* application, 18% of all activities in the *DS* application, and 20% of all activities in the *DS 100* application. An overview of the total number of activities and causes for events of interest, as well as the ratio of causes for events of interest to all activities for the three applications at the file level of granularity for artifacts of type *code* is shown in Table 6.5. Similar relationships between the different applications were observed at the logical level of granularity as well, although the ratios between the number of causes and the number of activities were lower at 10–13%.

6.4.2.1. Developer-Centric Models

To get an initial insight into how developer-specific models perform against project-specific models, we compared the three applications at the file level of granularity for artifacts of type *code* using the *DT* machine learning algorithm. When considering the three ways applying the machine learning algorithms (*All*, *DS*, and *DS 100*), there are a few issues to be discussed. The *DS* and *DS 100* applications rely on subsets of all the activities in *All*. As shown in Table 6.5, the difference between *DS* and *All* is smaller (6.9%), but the difference between *DS 100* and *All* is much larger (29.2%). To evaluate the impact of considering only the activities from the *DS* and *DS 100* applications in a project specific manner, we performed corresponding experiments, referred to as *All (DS)* and *All (DS 100)*. To obtain the activities for *All (DS)* and *All (DS 100)* we put together all the activities from *DS* and *DS 100* and split them into training and testing data the same way as in the *All* application.

The results from the experiments are summarised in Figure 6.15. For each application, we report the average F-measures for the causes for events of interest. Applying the algorithms in developer-specific manner (*DS*) in this scenario performs better than applying in project-specific manner (*All*). However, the *All* application also contains activities from small developers with less than 100 activities per developer. Removing the small developer from the project-specific application (*All (DS)*) performs similarly to *All*, indicating that the impact of the small developers can be considered to be negligible. Considering only very active developers for predictive modelling in developer-specific manner (*DS 100*) performs best in this case. On the other hand, considering only the

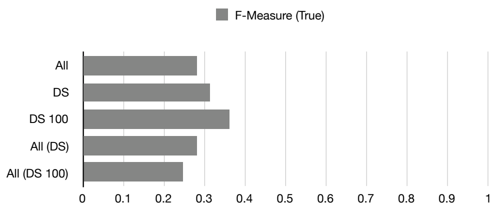


Figure 6.15.: F-measure comparison of different applications at the file (code) layer

activities of very active developers in a project-specific manner (*All (DS 100)*) performs even worse than *All*.

6.4.2.2. Developer-Centric Characteristics

To evaluate the impact of developer-centric characteristics, we considered three sets of characteristics at the different levels of granularity. We considered the file (*code*) and logical (*class* and *method*) levels of granularity. The default sets at each level of granularity include all characteristics available at the corresponding level. The *target* set includes only the characteristics at the target state of an activity, i.e. the source state characteristics as well as the deltas between the source and the target state are not used. The *target-core* set is a subset of the *target* set which only includes the core characteristics of the target state, excluding characteristics related to collaboration and experience. The relationships between the different sets of characteristics are summarised by means of a Venn-diagram in Figure 6.16

We performed experiments with all machine learning algorithms (*DT*, *LR*, *NB*, *RF*, *SVM*) considering the three sets of characteristics and the three levels of granularity, which produced nine results for each machine learning algorithm. The results from the experiments are summarised in Figure 6.17. For each machine learning algorithm, we report the F-measures for the causes for events of interest. We compare the different sets of characteristics at the different levels of granularity. The three ways applying the machine learning algorithms shown in different colors: *All* the average for all projects when all activities for each project are used; *DS* the average for all developers that have performed at least 100 activities; *DS 100* the average for all developers that have performed at least 100 activities and there are at least 100 events of interest in the training data of each developer. Several observations can be made based on Figure 6.17. In general, the results at the file level of granularity are better than the results at the logical levels of granularity. Depending on the algorithm, the additional characteristics may produce similar, or, at times, even worse results. *LR* and *SVM* in particular can still benefit from the additional characteristics, especially at the logical levels of granularity. Applying the algorithms in developer-specific way produces similar or better results in most cases. In particular, considering only developers having at least 100 events of

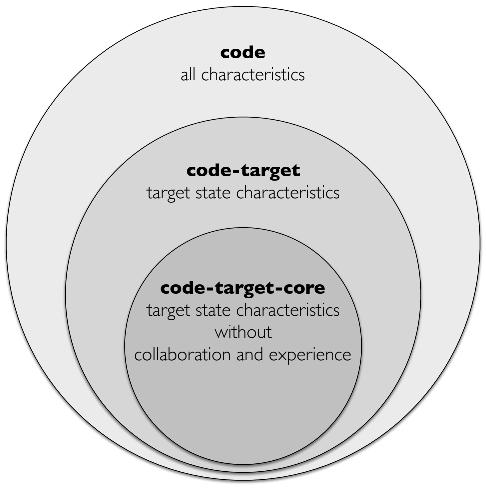

code
all characteristics
code-target
target state characteristics
code-target-core
target state characteristics
without
collaboration and experience

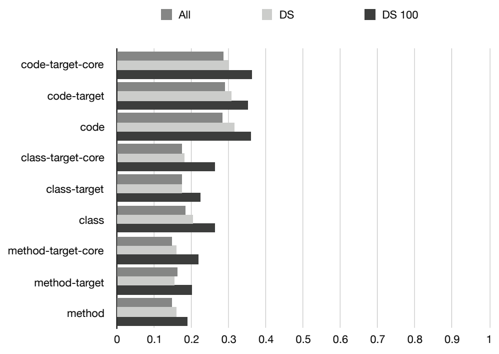
Figure 6.16.: Characteristics sets (code level of granularity)

interest in the training data seems to improve results even more substantially in most cases. This comes at the cost of excluding smaller developers. Grouping smaller developers together or applying the algorithms on the project as a whole could provide better results for activities performed by smaller developers.

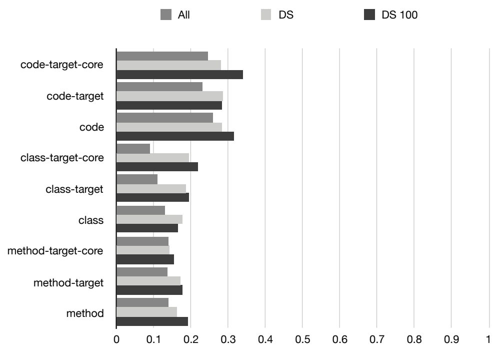
6.4.2.3. Thresholds for Small Developers

There have been reports in the literature related to the minimum number of activities for good results from which we can infer the threshold for small developers. Kim et al. [95] reported that their technique requires about 100 changes (activities) to *"train a project-specific classification model before the predictive accuracy achieves a "usable" level of accuracy"*. Jiang et al. [84] noted that their personalised technique perform better than an equivalent non-personalised technique when there are at least 80 (changes) activities per developer for training.

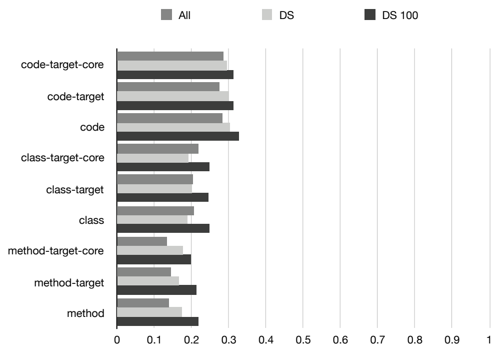
We evaluated different thresholds, both for the number of activities and for the number of events of interest. We evaluated the same machine learning algorithms (*DT*, *LR*, *NB*, *RF*, *SVM*) in a developer-specific manner against the data sets at the file (code) layer with true splitting and all characteristics included. We considered three different thresholds for the total number of activities for a developer: 100 activities (reported as *sd100*), 150 activities (reported as *sd150*), and 200 activities (reported as *sd200*). For these three thresholds, we also considered filtering the developers based on the number of events of interest in the training data, considering only developers that have at



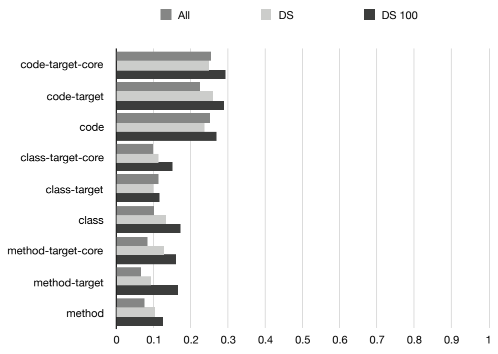
(a) Decision trees



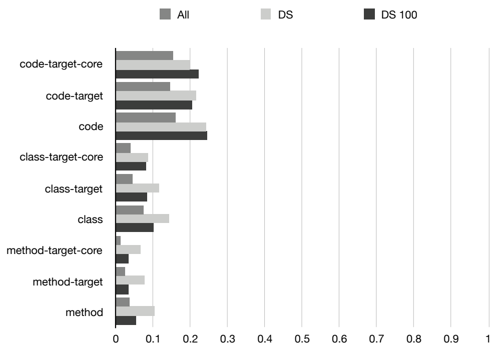
(b) Logistic regression



(c) Naïve Bayes



(d) Random forest



(e) Support vector machines

Figure 6.17.: F-measure evaluation for characteristics subsets at different layers

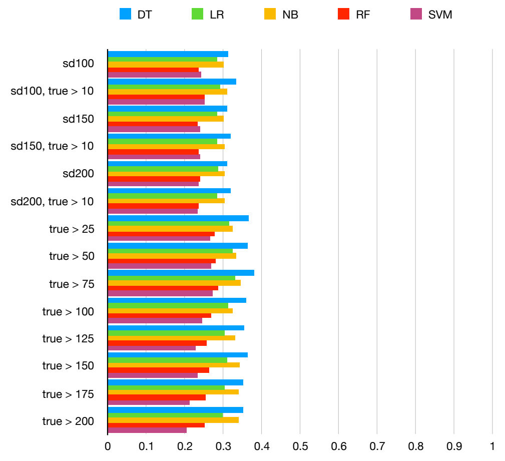


Figure 6.18.: F-measure comparison of thresholds for small developers at the file (code) layer

least 10 events of interest (reported as *sd100, true* > *10*, *sd150, true* > *10*, and *sd200, true* > *10*, respectively). Additionally, for the *sd100* threshold, we considered further thresholds for the minimum number of events of interest at increments of 25 between 0 and 200 events of interest, resulting eight additional thresholds for *sd100* (reported as *true* > *25*, *true* > *50*, *true* > *75*, *true* > *100*, *true* > *125*, *true* > *150*, *true* > *175*, *true* > *200*, respectively). The average results over all developers for the F-measure are summarised in Figure 6.18. We can observe that the thresholds based on the number of activities have little impact. Instead, the thresholds based number of events of interest lead to some improvements. Surprisingly, it is not necessarily the case that the results keep improving for larger thresholds based on the number of events of interest. There are some improvements for the F-measure for all algorithms at *true* > *25*, *true* > *50*, *true* > *75*, but beyond *true* > *100* the improvements stay flat or even decrease.

6.4.2.4. Small and Big Developers

Since predictive modelling for developers with fewer contributions typically does not perform well, we evaluated how grouping all the activities from such developers within a project (reported as *small*) compares to using all the activities within a project. Additionally, we also evaluated how well grouping all the activities from the developers with more contributions (reported as *big*) performs in the same context. Finally, we evaluated whether one group of activities can be used as training data for predicting the outcomes of activities in the other group.

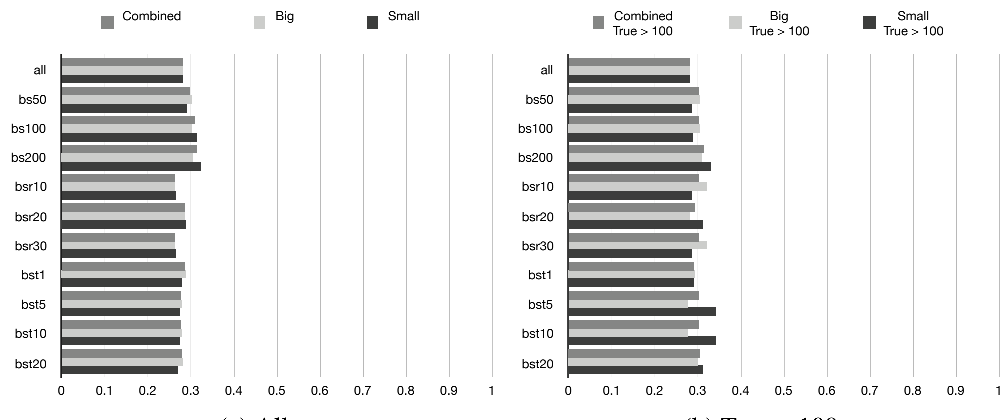
We considered three different ways of partitioning the developers within each project, based on the absolute number of activities for each developer within the project, whether a developer has performed more than a given percentage of all the activities within the project, and whether the developer ranks among the top *N* developers based on the total number of activities within the project. For the partitioning based on the number of activities, we used the thresholds at 50 (reported as *bs50*), 100 (reported as *bs100*) and 200 (reported as *bs200*). For the partitioning based on the percentage of activities, we used thresholds at 10% (reported as *bsr10*), 20% (reported as *bsr20*), and 30% (reported as *bsr30*). For the partitioning based on ranks, we used top 1 (reported as *bst1*), top 5 (reported as *bst5*), top 10 (reported as *bst10*), and top 20 (reported as *bst20*).

For the first experiment, we performed predictive modelling within the *small* and *big* partitions for each project at the file (code) layer using all characteristics and *TS*. We only used *DT* as algorithm in this scenario. We summarised the averages for the F-measure for each partition across the different ways of partitioning. In addition, we also included the averages from using both the *small* and *big* partitions (reported as *combined*). The results from this experiment are shown in Figure 6.19a. In general, there are small improvements over using all activities for a project without any partitioning (*all*), particularly when using threshold based on the number of activities of developers, where *bs100* and *bs200* yield the best results, especially for the *small* partitions. A closer look at the results revealed that the improvements are mainly due to increased precision for the *small*. Partitioning based on percentage of activities and ranks yield similar or even worse (*bsr10* and *bs30*) results compared to not using any partitioning.

Additionally, we also considered the scenario where only *small* and *big* partitions containing more than 100 events of interest (reported as *small true > 100*, *big true > 100*, and *combined true > 100*, respectively). The results from this experiment are shown in Figure 6.19b. There is overall increase compared to Figure 6.19a, especially in the partitioning based on percentage of activities and ranks, where the improvements are mostly in the averages for the *small* partitions.

For the second experiment, we performed transfer predictive modelling using the *small* partition to predict the *big* partition for each project (reported as *Small -> Big*) and the other way around (reported as *Big -> Small*) at the file (code) layer using all characteristics. In addition to *TS* within the partitions, since there is no overlapping between the *small* and *big* partitions, we also considered using the the complete partitions without any percentage splitting (reported as *Small -> Big NoPS* and *Big -> Small NoPS*, respectively). Finally, we considered a scenario where only *small* and *big* partitions containing more than 100 events of interest without percentage splitting are included (reported as *Small -> Big NoPS, True > 100* and *Big -> Small NoPS, True > 100*, respectively).

In this experiment we also used only *DT* as algorithm. We summarised the averages for the F-measures for both transfer scenarios in Figure 6.20. For both the *Big -> Small* and *Small -> Big* scenario, using *TS* generally yields comparable or worse results to not using any partitioning. Not using percentage splitting (*NoPS*) yields comparable or slightly improved results. Considering only partitions containing more than 100 events



(a) All (b) True > 100

Figure 6.19.: Partitioning developers into big and small according to different criteria at the file (code) layer

of interest yields best results with partitioning based on the number of activities again providing best option.

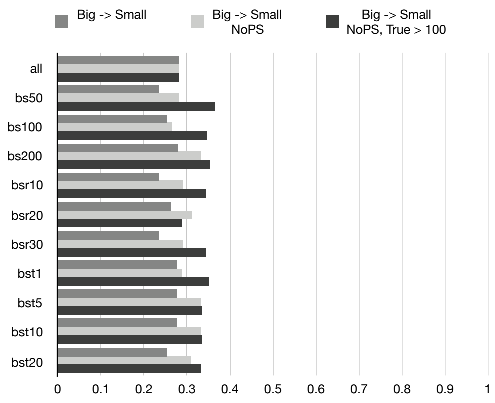
6.4.2.5. True Splitting

Due to the uneven distribution of the events of interest in time, in some of the experiments we applied the *TS* strategy for percentage splitting in order to ensure that the training and test data used for predictive modelling has the same number of events of interest. In a real-world application, it is unlikely that the operational data (for which the events of interest are not known) will include the same number of events of interest as the training data.

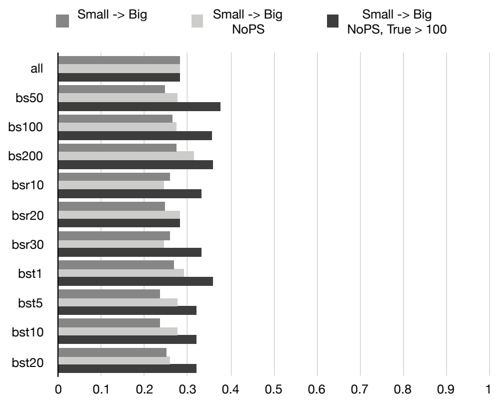
We performed an experiment to evaluate the impact of the *TS* strategy. We compared the averages of the F-measure results from predictive modelling using the three ways (*All*, *DS*, *DS 100*) of applying all algorithms ((*DT*, *LR*, *NB*, *RF*, *SVM*) for each project at the file (code) layer using all characteristics with and without *TS*. The results are summarised in Figure 6.21. We can observe that the impact of *TS* varies depending on the algorithm used. For *DT* there is very small difference, noticeable only when considering the *DS* application of the algorithm. For *LR*, *NB*, and *RF* there is a noticeable decrease in performance across all ways of applying the algorithms. However, for *SVM* there is even a noticeable improvement when *TS* is not used.

6.4.2.6. Undersampling

One of the ways to deal with bias in the data in predictive modelling is undersampling [42] (or subsampling), where all the data from the minority class is included and

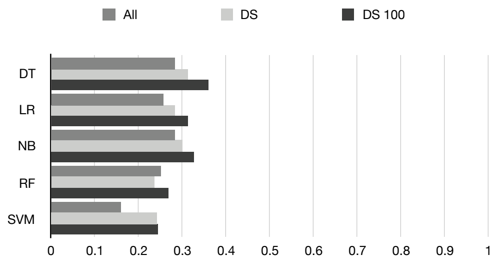


(a) Big -> Small

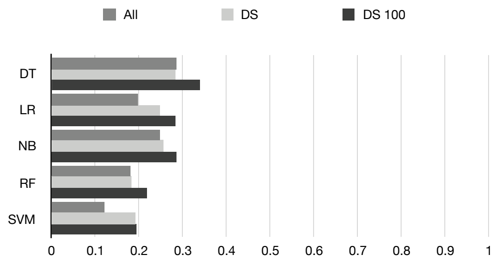


(b) Small -> Big

Figure 6.20.: Transferring between big and small developers at the file (code) layer



(a) With TS

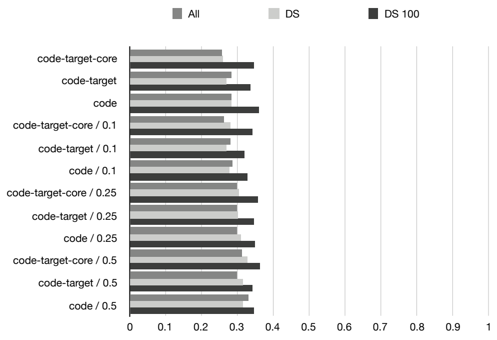


(b) Without TS

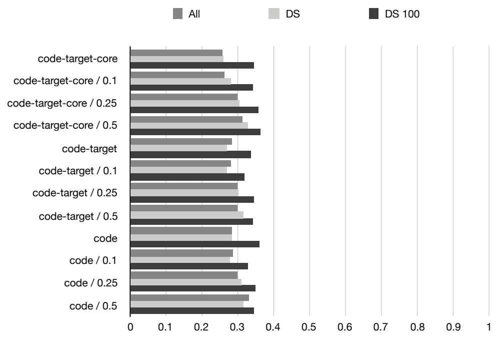
Figure 6.21.: F-measure comparison for percentage splitting at the file (code) layer

only a subset of the data from the majority class is used in order to reach a desired ratio between the different classes. In other words, with undersampling we attempt to obtain a specific ratio of events of interest and events of no interest.

To assess the impact of undersampling, we performed an experiment considering the different sets of characteristics (*target-core*, *target*, all) when applying the *DT* algorithm in the different ways (*All*, *DS*, *DS 100*) for all projects at the file (code) layer. We defined three different ratios — 0.1 where there is 1 event of interest per 10 activities, which is close to the original ratio of most projects, 0.25 where there is 1 event of interest for every 4 activities, which is close to the ratio of the projects with the highest ratios, and 0.5 where there is 1 event of interest for every 2 activities. We applied undersampling only to the training data (split without *TS*) which is more similar to a real-world application. Undersampling can only be performed to the training data since the events of interest in the test data are usually not known. The average F-Measures from the experiment are summarised in Figure 6.22. The baseline without



(a) Ordered by undersampling ratio



(b) Ordered by characteristics set

Figure 6.22.: F-measure comparison for undersampling at the file (code) layer

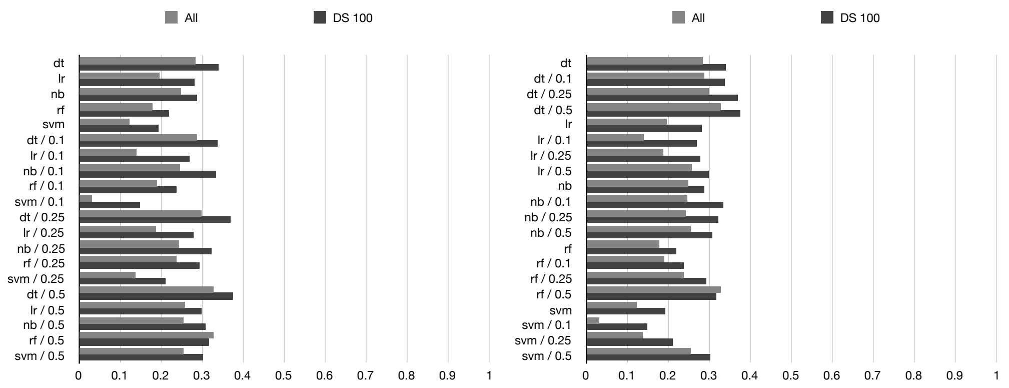
undersampling is shown at the top.

When we apply the undersampling, the F-measures for the *All* and *DS* are mostly similar and increase slightly with the increasing ratios. The baseline for *All* and *DS* performs similarly to using the 0.1 undersampling ratio. For the *DS 100* application, the F-measure stays largely the same, where at 0.1 there is even a small decrease. The *DS 100* application of the algorithm provides better results in all cases, however, the advantage of using it decreases as the undersampling ratio increases. The use of the different subsets of characteristics makes little difference, regardless of the undersampling ratio.

To investigate how the other algorithms perform with of undersampling, we performed an experiment using all characteristics at the file (code) layer. For brevity, we only report on the *All* and *DS 100* applications of the algorithms. The *DS* application usually scored somewhere between *All* and *DS 100*. We used the same three ratios (0.1, 0.25, and 0.5) and splitting without *TS*. The results for the F-measures averaged over all projects are summarised in Figure 6.23 with the baselines without undersampling shown on the top. As already noted for *DT* in the previous experiment, there is some increase in the performance as the ratio increases. For *LR* and *SVM* there are drops for *All* at 0.1, otherwise along *RF* they see the biggest gains as the ratio increases. For *NB*, the performance stays the same for *All*, but for *DS 100* there is even a slight decrease in performance as the ratio increases. *DS 100* is ahead in most cases with all algorithms, but the advantages diminish as the ratios increase, where for *RF* the *All* application performs slightly better than *DS 100* at 0.5.

6.4.2.7. Transfer Opportunities

In Section 4.3.6 we discussed different opportunities for applying transfer learning to developer-specific predictive models, both within the same project (*wp*) and across projects (*cp*). We consider two scenarios where we use the data from one developer



(a) Ordered by undersampling ratio

(b) Ordered by algorithm

Figure 6.23.: F-measure algorithm comparison for undersampling at the file (code) layer

to predict the outcomes of activities for the same developer (*wd*) and to predict the outcomes of activities for other developers (*cd*), both within the same project and across projects. This results in four scenarios (*wp-wd*, *wp-cd*, *cp-wd*, *cp-cd*).

We performed experiments with the *DT* algorithm at the file (code) layer, in order to get an initial evaluation of how the different scenarios compare to each other. We considered the three ways of applying the algorithms (*All*, *DS*, *DS 100*), where *All* serves as a baseline for which there is no difference between *wd* and *cd* since in this case there is no differentiation between developers. The results for the F-measures averaged over all projects are summarised in Figure 6.24.

Using transfer learning among developers within the same project (*wp-cd*) yields approximately the same results on average when applying *DS*, for developers with more contributions (*DS 100*) it performs slightly better. Across projects, using data for one developer from one project to predict the outcomes of activities for the same developer in another project (*cp-wd*) performs worse than using data for all activities of all developers (*cp* with *All*), on average. However, using data from one developer from one project to predict the outcomes of other developers in other projects (*cp-cd*) performs similarly to using data for all activities of all developers (*cp* with *All*), on average.

When considering the different possibilities for partitioning the activities of developers in groups for small and big developers in Section 6.4.2.4, we evaluated whether these add any benefit for transfer learning within the same project. In addition to transfer learning within the same project (*wp*), we evaluate transfer learning across projects (*cp*) when using the partitioning into small and big developers. We use the base scenario where developers are considered as small if they have less than 100 activities. We evaluated five scenarios, considering the combined effect of the small and big partitions (*bs-wp* and *bs-cp*), the small (*bs-wp-small* and *bs-cp-small*) and big (*bs-wp-big*

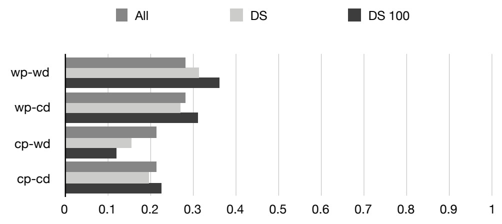


Figure 6.24.: F-measure comparison for transfer opportunities within and across projects at the file (code) layer

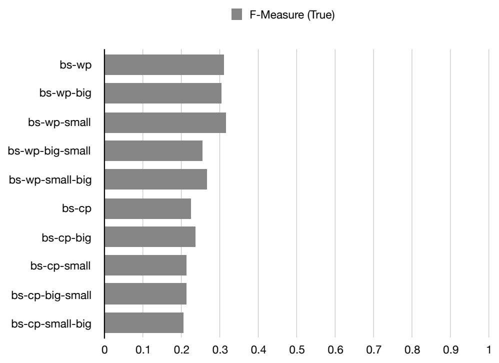


Figure 6.25.: Transfer for small and big developers at the file (code) layer

and *bs-cp-big*) partitions in isolation, using the big partition to predict the small partition (*bs-wp-big-small* and *bs-cp-big-small*), and using the small partition to predict the big partition (*bs-wp-small-big* and *bs-cp-small-big*).

We performed experiments only with the *DT* algorithm at the file (code) layer. The results for the F-measures averaged over all projects are summarised in Figure 6.25. Overall, the partitioning into small and big developers has little impact predictive modelling across projects. Using only the big partitions (*bs-cp-big*) can provide minor improvement on average when compared to using both small and big partitions, whereas the small partitions seem to have a negative effect on all other scenarios involving them (*bs-cp-small*, *bs-cp-small-big*, and *bs-cp-big-small*).

In Section 4.3.6 we also discussed the grouping of similar activities by means of clustering and different opportunities for applying transfer learning to developer-specific predictive models considering these groups (or clusters).

We consider scenarios where we use the data from one cluster to predict the outcomes of activities for the same cluster (*wc*) within the same project (*wp-wc*) and to predict the outcomes of activities for other clusters within the same project (*wp-cc*) as well as across projects (*cp-cc*).

We performed experiments only with the *DT* algorithm at the file (code) layer, in

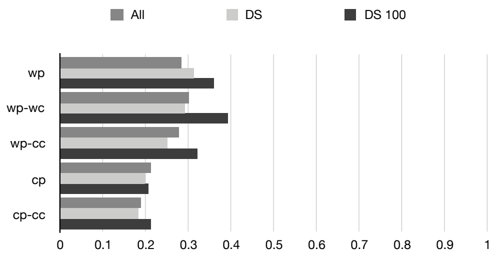


Figure 6.26.: Transfer for groups of similar activities at the file (code) layer

order to get an initial evaluation of how the different scenarios compare to each other. We considered the three ways of applying the algorithms (*All*, *DS*, *DS 100*), where for *All* we applied clustering to all activities from all developers within the project, whereas for *DS* and *DS 100* we applied clustering to the activities of each developer individually. The results for the F-measures averaged over all projects are summarised in Figure 6.26. Within the same project (*wp*), clustering all the activities for the project (*All*) yields comparable results for predictive modelling both within the same cluster (*wp-wc*) and across clusters (*wp-cc*). Across projects and across clusters (*cp-cc*) the results are also comparable but a bit worse than predictive modelling across project boundaries without clustering. Clustering the activities for each developer separately (*DS*) yields slightly worse results within the same project (*wp-wc*, *wp-cc*) on average, and comparable across projects (*cp-cc*. Considering only developers with more contributions (*DS 100*), there is a notable improvement within the same project and the same cluster (*wp-wc*), as well as across clusters (*wp-wc*).

7. Discussion

In this chapter, we discuss the results from the case studies and their interpretation with regard to the goals for the case studies and for the thesis as a whole, as well as results from evaluations in closely related work. Then, we discuss the strengths, limitations, and lessons learned from the work on this thesis. Finally, we discuss the identified threats to the validity of the results and findings in this thesis.

7.1. Results Interpretation

In this section, we summarise the findings from the case studies and reflect on them with regard to the goals for the case studies, and the thesis as a whole.

7.1.1. Identifying Potential Causes for Events of Interest

To asses the impact of the multi-layer approach, we investigated the potential benefits of using the multi-layer weighting approach at finer levels of granularity and evaluated the impact of using different weight distribution strategies. When comparing the multi-layer approach to simply inheriting labels from containing artifacts, the multi-layer approach achieved a reduction in the number of potential causes for events of interest between 50% and 80% at the file layer. At the logical layer, the reduction was between 75% and 95%. Consequently, the multi-layer approach can be applied to effectively reduce the amount of noise at finer levels of granularity when identifying potential causes for events of interest.

Considering weights rather than binary classification adds a further refinement, where causes for events can be prioritised based on the confidence indicated by the weight. In the case studies, we observed that the application of the weighted multi-layer approach generally results in a lower number of low confidence causes and at the same time in a higher number of high and medium confidence causes, both of which are desirable outcomes of the approach. However, the distribution of the weights may still vary between projects depending on the nature of changes recorded in the VCS.

During the application of weight distribution strategies, we observed that all strategies have largely similar effect on lower confidence causes. The effects on higher confidence causes vary between projects and levels of granularity, which is a desirable outcome as different strategies are intended to highlight different aspects of fixes for causes of events of interest. Overall, the use of weight distribution strategies provides a more nuanced view enabling prioritisation based on certain characteristics of the changes and the context in which they occurred.

Overall, the results from the case studies demonstrated the application of the approach for the identification of potential causes for events of interest and its benefits for obtaining more refined data and filtering out noise at finer levels of granularity. For the identification of potential causes for events of interest, we focused only on one kind of events of interest, namely bug-fixes, as identified by the *BugFix* factor. The applicability of the approach to other kinds of events of interest remains to be investigated in future work.

7.1.2. Developer-Centric Assessment

We proposed a more comprehensive way to characterise developer behaviour based on situational and dispositional characteristics related to the context in which an activity occurs. To assess the impact of the different characteristics, we performed experiments considering three sets of characteristics, including all characteristics, only characteristics related to the target state of an activity as well as only characteristics related to the target state of an activity, excluding characteristics related to collaboration and experience. The experiments indicated that the additional characteristics have little impact on the predictive models with some benefits for *LR* and *SVM*, but no or even negative impact for other machine learning algorithms.

Applying predictive modelling in a developer-specific way produces similar or better results in most cases. In particular, for active developers having at least 100 events of interest in the training data, there is larger improvement in most cases, but this comes at the cost of excluding smaller developers. The latter can be addressed by applying predictive modelling to all smaller developers together or using a project-specific predictive model, both of which performed similarly in our experiments. While the threshold was initially selected based on related notes from the literature [95, 84], we also considered other thresholds, both with respect to the number of activities and with respect to the number of events of interest. The thresholds based on the number of activities had little impact, whereas the thresholds based number of events of interest lead to some improvements. There were some improvements for all algorithms when considering only developers having at least 25, 50, 75, and to a lesser extent 100 events of interest. Above 100 events of interest there was even a slight decrease.

We observed limited opportunities for transferring developer-specific predictive models from one developer to another, which varied depending on the algorithm being used. In particular, predictive models for the same developer transferred across different projects performed worse on average when compared to generic project-specific predictive models transferred across different projects. In general, predictive models for developers transferred across different projects performed similarly or better when compared to generic project-specific predictive models transferred across different projects.

We made similar observations on the transfer opportunities for predictive models based groups of similar activities. On average, using predictive models on groups of similar activities has some benefits for active developers. However, transferring predictive models between different groups of similar activities performs worse than developer-specific predictive models (but still better than generic project-specific predictive models). Across projects, grouping of similar activities for predictive modelling yielded no benefit.

Besides the grouping of similar activities, changes in the behaviour of developers were explored only qualitatively to a limited extent in Section 4.3.1 with respect to the changes in the importance of the different characteristics. Other experiments indicated that there is some variability, however, we did not observe any specific patterns. Further studies shall be performed to quantify the attribute stability and investigate correlations between the attribute stability and the changes in the reliability of predictive models more systematically.

7.1.3. Reflections

In the following, we reflect on the ways the work on this thesis contributed to answering the research questions defined in Chapter 1.

Regarding the first research question, "*How can we characterise developer behaviour based on information collected from software-related assets?*", we described a comprehensive approach for the characterisation of developer behaviour seeking to characterise the circumstances in which development activities are performed. The conceptual approach is based on the notions of situational and dispositional factors, as well as collaborative factors. Characteristics across different dimensions can be used to capture the context and outcome each activity. While we considered only information that can be automatically extracted from software-related assets, further information collected by other means, such as questionnaires and interviews, can be integrated in the conceptual model as well. Visualisation and data mining techniques can be used to gain further insights based on the resulting data. We discussed six different approaches which can be applied to support decision making during software development.

Regarding the second research question, "*How can we determine potential causes for events of interest across multiple levels of abstraction?*", we described an approach adds quantitative information on top of existing approaches for origin analysis. The quantitative information in the form of weights can be calculated independently for different kinds of events of interest, such as bug fixes, refactorings, etc., and distribution across multiple levels of granularity. This way, we can obtain more accurate information regarding the likely causes for the events of interest at finer levels of granularity. The different strategies for weight redistribution can emphasise different aspects, such as the size of the affected artifacts and the amount of change they have undergone, in order to account for the importance of these aspects when prioritising quality assurance efforts.

The corresponding case studies demonstrated the application of the approach its benefits for obtaining more refined data and filtering out noise at finer levels of granularity.

Regarding the third research question, “*How can we mine information related to developer behaviour and its impact on potential causes for events of interest from software-related assets in an effective and agile manner?*”, we described a model-based software mining approach which is based on domain-specific meta-models. The approach can serve as a glue for loosely coupling different existing third-party and custom-made software mining solutions at a high level of abstraction, easing the integration of diverse and heterogeneous assets. As a proof of concept, a concrete instantiation of the approach was used for the realisation the approaches for identification of potential causes for events of interest and characterising developer behaviour. It demonstrated the integration of various tools which are widely used in research, providing a convenient solution for the integration of various input and output formats.

Regarding the fourth research question, “*What are the advantages of using developer-centric software assessment?*”, in addition to providing more detailed and personalised information, the developer-centric approach has the potential to provide more accurate predictive models. We performed experiments to assess the impact of the different characteristics, the application of predictive modelling in a developer-specific way, as well as transfer opportunities for developer-specific predictive models. The experiments indicated that the additional characteristics have little impact on the predictive models. On the other hand, applying predictive modelling in a developer-specific way produces similar or better results in most cases, particularly for very active developers. We observed limited opportunities for transferring developer-specific predictive models from one developer to another. Predictive models for the same developer performed worse across different projects, but predictive models for different developers performed similarly or better across different projects when compared to generic project-specific predictive models across different projects.

7.2. Comparison to Related Work

With respect to the approach for the identification of potential causes for events of interest, to the best of our knowledge there are no other approaches incorporating the quantification of the extent to which a change in one state contributes to a subsequent fix in a later state of an artifact, in particular also how to apply such quantification of the cause-fix analysis across multiple levels of granularity.

In terms of developer-centric assessment, multiple approaches have pursued the integration of various developer-related characteristics with varying degrees of success. In particular, two recent contributions [84, 163] evaluated developer-specific defect prediction models.

Shihab et al. [163] focused on risky rather than buggy changes in an industrial setting. Changes and commits are used interchangeably in their work, corresponding to an activity / state at the project level of granularity in this thesis. They achieved an overall improvement by using developer-specific predictive models, however, they reported their results in an unusual manner, making them hard to compare. In addition, they used different characteristics and performed cross-validation rather than percentage splitting. They reported the "predictive and explanative power results for the top 10 developers" [163] (based on the total number of changes) where they report 87% improvement in precision over the baseline model, while achieving an average recall of 0.677. The baseline model they considered is "a model that randomly predicts risky changes" [163]. The selection criteria for the developers they considered are also different, where they "selected developers who made at least 20 changes over the year studied" while requiring "that at least 20% of a developer's changes belong to either class, risky or non-risky" [163].

Jiang et al. [84] evaluated predictive models for individual developers and observed significant improvements when using personalised change classification. They evaluated their approach on six large open source projects, none of which are included in the data set for this thesis, and they also used different characteristics, including characteristic vectors. They used an unusual approach to data selection where they used arbitrary gaps in the beginning and end of the recorded periods for the selected projects. The authors also selected only the top 10 developers from each project who have the most commits. Note that the authors refer to their approach as a change classification problem, where "A change is the lines modified in one file of a software version control system commit." [84], which would correspond to an activity / state at the file level of granularity in this thesis. It is unclear why they select the top 10 developers based on the number of commits. For the baseline they picked "the same number of changes (100) from each of the developers to prevent any developer's performance from dominating" [84], where it is not clear whether they really distinguish between commits and changes. Finally, they also performed cross-validation. In their results, they observed some minor improvement on average in terms F-measure (0.03), in particular in their enhanced approach considering a majority vote meta-classifier combining the outcomes from the baseline and developer-specific predictive models. They also noted that the improvements were not limited to a specific machine learning algorithm (although there is some variation in the results reported in their work).

More recently, Xia et al. [189] followed up on the work of Jiang et al. [84] by leveraging a multiobjective genetic algorithm to combine predictive models for different developers with different weights. They evaluated their approach on the same six open source projects as Jiang et al., using a similar setup and assumptions. In their results, they observed some minor improvement on average in terms F-measure with respect to the results from Jiang et al. (0.01 and 0.02).

7.3. Strengths and Limitations

In this thesis, we proposed three approaches to address challenges in software assessment and in particular developer-centric software assessment. For every approach there are corresponding strengths and limitations. In the following sections we discuss these from a pragmatic point of view, as well as the lessons we learned along the way.

7.3.1. Events of Interest... or No Interest

In the approach for the identification of events of interest and their causes we had to make certain assumptions that are also commonly relied upon in the literature. However, there are also concerns regarding the quality of the data found in software repositories, especially for open source projects where there may be no clear conventions regarding development practices and traces, or if there are, they may not be strictly enforced. Thus, the reliability of the identified events of interest hinges upon the consistent use of certain keywords or other indicators for events of interest, which may also vary from project to project. The approach discussed in this thesis is decoupled from the exact way of identifying events of interest which serves as a foundation. Thus, more refined and reliable approaches for the identification of events of interest can be used as a foundation in order to improve the overall accuracy of the results.

We focused our investigation on only one kind of event of interest — *bug fixing* which is commonly used in the literature and has a potentially high impact on practitioners. Other kinds of events of interest, such as refactorings and reductions in technical debt can yield different results. The suitability of the approach for these kinds of events of interest remains to be investigated.

While in theory changes related to different events of interest shall be neatly separated, in practice, these are often tangled together [76]. A bug fix and a refactoring, which may or may not be related to the bug fix, may be recorded as a single change in a VCS. The strategies for weight distribution discussed in this thesis can offset the impact of tangled changes to an extent. A more targeted approach for untangling changes, as discussed in [76], may be necessary to obtain more accurate information regarding the different kinds of events of interest.

The different strategies for weight distribution discussed in this thesis are rather generic. Selecting the appropriate strategy depends, in part, on the intended application context. Further strategies may be defined to target additional application contexts. Even with the application of adequate strategies for weight distribution, noise may still have a considerable impact. The application of the strategies seeks to aid rather than replace careful data inspection and cleansing. Visualisations based on the strategies may be helpful for understanding the impact of the strategies, but automated heuristics and filtering are required for application at scale.

7.3.2. On the Shoulders of Giants... or Dwarves

Relying on third-party tools can be both a blessing and a curse. Due to the scope of the work needed for this thesis, we had to rely on a number of existing third-party tools and integrate them to produce the data for the case studies.

In general, the use of third-party tools can lower the barrier to entry by providing a quick access to a multitude of facts, however, they shall be evaluated with caution. While the intuition was to benefit from upstream development and reuse existing capabilities, there are a number of challenges related to the (over-) reliance on third-party tools. First, every tool has been designed for certain usage scenarios envisioned by its creators and/or users. Consequently, there are inherent limitations to how a tool can be used beyond the scenarios which it was originally designed for. In addition, the tools themselves may have inherent problems and fail to provide accurate results in certain circumstances. While open source tools can be customised and fixed in theory, in practice this often incurs considerable overhead. Commercial tools on the other hand can only be used as they are provided. If there are problems, they can be reported to the tool vendor. However, depending on the priority of the problems, it may take a while before they are addressed or they may not be addressed at all.

Despite careful consideration and selection of sustainable tools, there is a risk that tools may cease to be available or compatible during the course of a project. This concerns both open source and commercial tools. Open source tools may no longer be maintained or acquired by a commercial organisation. Commercial tool vendors may stop producing a tool or even run out of business. It is hard to foresee whether the tools will continue to be available and compatible in the future. Therefore, it shall be easy to integrate new tools and substitute existing tools with as little overhead as possible.

The model-based approach seeks to alleviate emerging problems to an extent in that makes it easy to replace tools and add new ones. As new tools emerge, they may become attractive for a variety of reasons, but it also may become a necessity to switch at some point. During the course of this thesis, we switched between three different tools for static analysis, and two different tools for duplicate detection. An open source test coverage tool that was considered early on turned into commercial tool and received no further maintenance after a certain point.

While the model-based prototype provided a foundation for further integration, the next frontier is scaling it up and moving to the cloud. With that step, new challenges emerge, as the facilities available from the cloud provider need to be considered as well. Due to the complexity of the mining tasks, testing and debugging can be very challenging. Moving to the cloud, this becomes even more challenging and also even more important. Deployment environments need to be replicated, otherwise differences in deployment targets may add further overhead. Ultimately, deployment and testing shall be integrated and streamlined to facilitate quick turn around during development of new assessment tasks.

7.3.3. Scaling Up… or Down

We collected and evaluated data from 16 projects of varying sizes and durations. The data extraction, transformation, and processing was performed directly on the raw assets and was a very computationally- and data- intensive process. While the required level of detail implies working with large amounts of data as well as corresponding computational demand, the operational data from the third-party tools used to extract facts did account for considerable overhead both in terms of required storage space and in terms of additional processing required to filter and transform this data into models and integrate the models from the different sources into the assessment model. Additionally, third-party tools incur considerable overhead for their repeated execution where initialisation steps have a high impact on the total runtime of the facts extraction step. Alternative tooling can lead to substantial improvements in both aspects, however, tools providing additional or different facts may also come at an increased cost in terms of computational demand.

Working with large models can be challenging. Several approaches [7, 10, 36, 159] seek to address different aspects of dealing with very large models. Due to their focus, they do come with some limitations and while initial benchmarks with the different prototypes have shown some promise, they are not yet mature enough for generic large-scale deployment. In our experiments, we did run into some limitations for the selected technologies, both in terms of required memory, runtime, and parallelisation opportunities. We explored some alternative solutions but they were not suitable at the time. Further investigations are necessary to evaluate their suitability as these solutions become more mature. Currently, assessment models for very larger projects do require substantial amounts of time and memory to process and use.

In addition, more sophisticated analytics can be quite demanding as well. Running experiments with *SVM*, for example, took several hours and in some cases also several days. Some visualisations can also take a while to be computed. More scalable and distributed approaches can improve the processing times where applicable.

With the lessons learned during the course of the work on this thesis, new cloud-based integrated software mining and assessment platform emerged to scale up the mining efforts further. The initial version of the SmartShark [174, 175] platform integrated the model-based software mining infrastructure proposed in this thesis. A newer version of the SmartShark platform addresses some of the limitations identified during the initial experiments and relies on custom facts extraction tools and a more scalable approach for all tasks related to software assessment.

Mining at a massive scale presents new opportunities for more refined experiments. Considering more projects and especially more diverse projects can help to validate the conclusions from this thesis and also yield new insights. More projects involving similar or identical developers can present further transfer opportunities. Considering further sources of information as well as additional facts enables new assessment tasks and can also improve the outcomes of the assessment tasks and approaches discussed in this thesis. However, adding more data and especially more diverse data increases the inherent complexity of the integration tasks and demands further validation. Finally, deploying developer-centric software assessment in an industrial software development setting may yield new insights in comparison to open source software development.

7.4. Threats to Validity

As with all empirical studies, there are certain threats to the validity of the findings from the case studies discussed in this thesis. In the following sections we discuss the threats we identified with regard to the internal, external, and construct validity of the findings.

7.4.1. Internal Validity

The threats to the internal validity are concerned with systematic bias in the conclusions with regard to the data and methods being used.

With respect to the identification of causes for events of interest and the evaluation of the strategies for weight distribution, the outcomes depend on the quality of the underlying data and the assumptions about the importance of different aspects related to the events of interest. The ground truth difficult to establish without extensive manual inspection, which can in turn be used to determine the best suited strategy for the weight distribution. Tangled changes in particular may also have a substantial impact on the outcome of the identification of causes for events of interest. Untangling changes [76, 39] can be an option, but, depending on the underlying data, it may still not be reliable enough. It is also not readily available in practice.

With respect to the overall quality of the tools being used and resulting data, the dependence on third-party tools can result in flawed data, but on the other hand, given the wider user-base it is less likely that issues would go unnoticed. Nonetheless, automated data validation is essential, given the size and scale of the data being collected. During the data collection, we did perform semi-automated validation at the initial stages, however, fully automated validation is still needed to rule out any issues. We relied on the same tools for all projects, thus, any unnoticed flaws would affect all the studied projects equally. Still, even with the same tool, the quality of the data may also vary between different supported languages. To this end, the logical level studies are done largely as proof of concept at this point and have been excluded from some of the results until further validation can be performed. A comparison with other tools would also be helpful to identify outstanding discrepancies.

With respect to the selected techniques, the approaches discussed in this thesis describe fundamental shifts in perspective. We illustrated their applicability with a selection of techniques, including directed and undirected data mining techniques, data visualisation, model-based software development, weight- and factor- based identification of potential causes of events of interest. The case studies performed in this thesis serve as an initial indication of the potential of the described approaches. The full extent of practical applicability is still subject to further studies. Due to the modularity of the proposed approaches, different constituents of the selected techniques may be substituted with more sophisticated alternatives as far as these are available.

With respect to the data used for the prediction of causes for events of interest, as is quite common in publicly available data sets, and observed for software projects in general, the artifacts containing defects and the defect-related activities comprise a rather small proportion of the overall data set. This leads to an inherent imbalance in the data used for predictive modelling. One of the approaches to deal with such data sets is undersampling and we investigated its impact on the overall results. The range of selected projects in terms of size and proportion of defect-related activities is similar to other studies reported in the literature, however, it is by no means representative of all the different kinds of software projects.

7.4.2. External Validity

The threats to the external validity are concerned with the generalisation of the conclusions beyond the specific data being used. We selected at random 16 open source projects using two different programming languages (Java and C++) spanning different domains and different ecosystems. The varied project sizes and durations can be considered representative for other projects. Still, given the vast number of open source projects and broad diversity of languages and development approaches being used, the results may not generalise beyond the selected languages, domains, and ecosystems, or even within them. We selected projects at random for a pilot study, a more systematic approach to project selection may be beneficial for further investigations. A scaled up investigation with supporting cloud platform could yield a more comprehensive understanding of the benefits and drawbacks of the presented approaches.

Regarding the approach for the identification of causes for events of interest and the different strategies, the quality of the results depends on the quality of the data available in the repositories. The results on other data sets may be better or worse depending on the quality of those data sets.

With respect to the developer-centric modelling, depending on the project size and work distribution among the developers, the outcomes may vary. While we selected projects exhibiting different patterns in the contribution behaviour of developers, different circumstances such as different development approaches may lead to other patterns in the contribution behaviour of developers.

When mining software repositories at a large scale, considering the vast amounts of data collected and the various tools involved in the extraction of data, other tools for other languages or additional measurements may yield different results.

7.4.3. Construct Validity

The threats to the construct validity are concerned with the suitability of the chosen measurements with regard to the conclusions. To assess the overall suitability of the tools and techniques being used, we performed manual inspection and isolated testing, using both constructed examples and real-world data for validation and exploration.

During identification of causes for events of interest we rely on reasonable and consistent recording of reasons for changes in VCSs. The weight-based approach seeks to quantify the contribution of each change to causing a particular event of interest. In practice, the recording of the reasons for changes may not always be consistent, especially in open source projects. In addition, changes affecting multiple artifacts, such as unrelated or tangled changes, dilute the weight of each individual change. While this is intended by design in the proposed approach, and the weight distribution strategies can be used for further refinement to allow focusing on larger modifications or larger artifacts affected by a change, further filtering, such as excluding certain types of modifications can yield more accurate results. Manual analyses were performed at a small scale to check the validity of assumptions. Considering the scale of the studies and the diversity of the studied projects, especially in cases where a project has undergone a switch of VCS, ITS, or other infrastructure component, further investigations shall be performed to asses the impact of such changes. Furthermore, a shift in development strategy, which is common as projects mature, can have similar consequences, where assumptions about the way how information is recorded in the VCS and/or the ITS may need to be adapted over time.

The selection of 16 random open source projects of different sizes, using two different programming languages, spanning different domains and different ecosystems, seeks to reduce the impact of peculiarities in individual projects. However, a larger selection of even more diverse projects is needed to assess the impact of the size, domain, language, and other aspects, in order to derive more refined conclusions specific to certain groups of projects as well as also universally applicable conclusions.

8. Conclusions

In this last chapter, we summarise the overall findings and contributions of this thesis, as well as draft ideas for future work on extending and refining the research presented in this thesis.

8.1. Summary

The overarching goal of this thesis is to investigate means for characterising developer contribution behaviour and assessing its impact on the resulting software products with respect to certain events of interest. The characterisation and assessment is based on traces collected from different kinds of software-related assets, containing information related to software artifacts at different levels of granularity. Pursuing this goal, we made several contributions within the scope of this thesis, which are related to the identification of potential causes for events of interest and the characterisation of developer behaviour, as well as a model-based approach for mining software repositories and conducting software assessment. We performed case studies to evaluate the methods described in this thesis.

The approach for the identification of potential causes for events of interest adds quantitative information on top of existing approaches for origin analysis. The quantitative information in the form of weights can be calculated independently for different kinds of events of interest, such as bug fixes, refactorings, etc. The approach accommodates weight redistribution across multiple levels of granularity in order to provide more accurate information regarding the likely causes for the events of interest. We outlined different strategies for weight redistribution, which emphasise different aspects, such as the size of the affected artifacts and the amount of change they have undergone, in order to account for the importance of these aspects. While there are different related approaches described in the literature, none of them incorporate quantification, especially across multiple levels of granularity. The present approach builds on top of these approaches where any of them can serve as a foundation, upon which quantitative information can be added at the various levels of granularity. The corresponding case studies demonstrated the application of the approach its benefits for obtaining more refined data and filtering out noise at finer levels of granularity.

The approach for the characterisation of developer behaviour is based on the notions of situational and dispositional factors, as well as collaborative factors, all of which seek to characterise the circumstances in which development activities are performed. We presented a selection of characteristics across different dimensions and discussed different approaches for making use of the resulting data based on visualisation and data mining techniques. While the conceptual approach to characterising developer behaviour is novel, many of the individual characteristics have already been discussed in the literature. The generic nature of the approach permits the use of other sets of characteristics beyond the ones discussed in this thesis, effectively also enabling the systematic description of the behaviour of any kind of entities performing activities on any kind of artifacts, beyond the domain of software engineering. In the corresponding case studies, we performed experiments to assess the impact of the different characteristics, the application of predictive modelling in a developer-specific way, as well as transfer opportunities for developer-specific predictive models. The experiments indicated that the additional characteristics have little impact on the predictive models. On the other hand, applying predictive modelling in a developer-specific way produces similar or better results in most cases, particularly for very active developers. We observed limited opportunities for transferring developer-specific predictive models from one developer to another. Predictive models for the same developer performed worse across different projects, but predictive models for different developers performed similarly or better across different projects when compared to generic project-specific predictive models across different projects.

The model-based software mining approach is based on domain-specific metamodels related to both the assessment tasks and the facts extracted by third-party tools aiming to ease the integration of the typically diverse and heterogeneous facts assets. The integration is achieved by means of model transformations in a stepwise manner. The integrated assessment model can be queried to produce assessment assets which used by third-party assessment applications. The approach serves as a glue for loosely coupling different existing third-party and custom-made software mining solutions at a high level of abstraction. A concrete instantiation of the approach was used for the realisation the approaches for identification of potential causes for events of interest and characterising developer behaviour. For the instantiation we relied on various tools which are widely used in research. The model-based approach was a convenient solution for obtaining high-level models from the various lower-level representations. This made the integration of various input and output formats very convenient and reusable. The proposed approach can lower the barrier to entry for researchers and practitioners by allowing them to focus on the assessment tasks rather than the technicalities.

8.2. Outlook

The work on this thesis explored several novel areas of research. While it advanced the state of the art in these areas, there are also numerous opportunities for further research to address open questions, refine the described approaches, and apply the approaches in further contexts under different circumstances.

For the identification of potential causes for events of interest, we focused only on one kind of events of interest, namely bug-fixes. The suitability of the approach for other kinds of events of interest such as refactorings needs to be investigated in future work. In addition, the overall approach can benefit from more refined and reliable approaches for the identification and untangling of events of interest which serve as a foundation for the approach discussed in this thesis.

Changes in the behaviour of developers were explored only to a limited extent. Further studies shall be performed to investigate the variability of developer-specific predictive models, as well as possible correlations between the stability of attributes and the reliability of predictive models more systematically.

Despite the benefits of using a model-based approach to software mining, working with large models can be challenging. Several approaches seek to address different aspects of dealing with very large models. Initial evaluation indicated that they were not suitable for our needs. Further investigations are necessary to evaluate their suitability as these solutions become more mature. To support larger scale software mining and software assessment, a deployment in a cloud environment is of particular interest for future work. We have started exploring this scenario within a cloud-based smart data platform for supporting empirical software research at a massive scale. Considering more projects can help to validate the conclusions from this thesis and also yield new insights. Further sources of information as well as additional facts can enable new assessment tasks and can also improve the outcomes of the assessment tasks and approaches discussed in this thesis. However, adding more data and especially more diverse data increases the inherent complexity of the integration tasks and demands further validation.

The investigations in this thesis are primarily concerned with object-oriented open source software implemented in the Java and C++ programming languages. The overall methodology can be adapted to other programming paradigms and programming languages. We rely on open source software due to the wide availability of publicly accessible information related to software projects from very diverse domains involving developers of various backgrounds from around the world. Subsequent case studies may apply the methodology to industrial software development and compare the findings.

While the work on this thesis sparked interesting academic discussions, ultimately, the intention is to also transfer the approaches into practice and allow the software developers unwittingly serving as subjects for various studies to benefit from the findings in these studies. Directing targeted feedback to the developers who are most likely to benefit from it, rather than producing generic feedback that may or may not apply to a large part of the developer population, can make the software development process more efficient. Complementing the quantified understanding that developer can gain regarding their own strengths and weaknesses, forward-looking organisations shall embrace a better understanding of their developers and teams and use this understanding to guide organisational strategies in order to maximise productivity and job satisfaction of developers. By looking at the strengths and weaknesses of each developer, organisations can invest in reinforcing these strengths and taking actions to compensate for or reduce known weaknesses. This can be achieved by incentivising and optimising development activities, implementing targeted organisational quality assurance measures where applicable, and organising targeted training to boost certain skills of the developers.

‘When I look back over the last 25 years, in some ways what seems most precious is not what we have made but how we have made it and what we have learned as a consequence of that. I always think that there are two products at the end of a programme; there is the physical product or the service, the thing that you have managed to make, and then there is all that you have learned. The power of what you have learned enables you to do the next thing and it enables you to do the next thing better.’[42]

— Sir Jonathan Ive

[42]From `https://www.wallpaper.com/design/jony-ive-apple-park`

Bibliography

[1] A. J. Albrecht and J. E. Gaffney. Software function, source lines of code, and development effort prediction: A software science validation. *IEEE Trans. Softw. Eng.*, 9(6):639–648, Nov. 1983.

[2] S. W. Ambler. *Agile Modeling: Effective Practices for Extreme Programming and the Unified Process*. John Wiley & Sons, Inc., New York, NY, USA, 2002.

[3] E. Arisholm and L. C. Briand. Predicting fault-prone components in a java legacy system. In *Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*, pages 8–17, Rio de Janeiro, Brazil, 2006. ACM.

[4] E. Arisholm, L. C. Briand, and E. B. Johannessen. A systematic and comprehensive investigation of methods to build and evaluate fault prediction models. *J. Syst. Softw.*, 83(1):2–17, 2010.

[5] D. Arthur and S. Vassilvitskii. k-means++: the advantages of carefull seeding. In *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 1027–1035, 2007.

[6] M. Balint, T. Girba, and R. Marinescu. How Developers Copy. In *Program Comprehension, 2006. ICPC 2006. 14th IEEE International Conference on*, pages 56–68, 2006.

[7] K. Barmpis and D. S. Kolovos. Comparative Analysis of Data Persistence Technologies for Large-scale Models. In *Proceedings of the 2012 Extreme Modeling Workshop*, XM '12, pages 33–38, New York, NY, USA, 2012. ACM.

[8] V. R. Basili, G. Caldiera, and H. D. Rombach. The goal question metric approach. In *Encyclopedia of Software Engineering*. Wiley, 1994.

[9] I. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *, International Conference on Software Maintenance, 1998. Proceedings*, pages 368–377, Nov. 1998.

[10] A. Benelallam, A. Gómez, G. Sunyé, M. Tisi, and D. Launay. Neo4emf, A Scalable Persistence Layer for EMF Models. In *Proceedings of the 10th European Conference on Modelling Foundations and Applications - Volume 8569*, pages 230–241, New York, NY, USA, 2014. Springer-Verlag New York, Inc.

[11] H. Benestad, B. Anda, and E. Arisholm. Understanding cost drivers of software evolution: a quantitative and qualitative investigation of change effort in two evolving software systems. *Empirical Software Engineering*, 15(2):166–203, Apr. 2010.

[12] A. Bernstein, J. Ekanayake, and M. Pinzger. Improving defect prediction using temporal features and non linear models. In *Ninth international workshop on Principles of software evolution: in conjunction with the 6th ESEC/FSE joint meeting*, IWPSE '07, pages 11–18, New York, NY, USA, 2007. ACM.

[13] J. Bezivin and O. Gerbe. Towards a precise definition of the OMG/MDA framework. In *16th Annual International Conference on Automated Software Engineering, 2001. (ASE 2001). Proceedings*, pages 273–280, 2001.

[14] B. Biegel, Q. D. Soetens, W. Hornig, S. Diehl, and S. Demeyer. Comparison of similarity metrics for refactoring detection. In *Proceeding of the 8th working conference on Mining software repositories*, MSR '11, pages 53–62, New York, NY, USA, 2011. ACM.

[15] C. Bird, N. Nagappan, B. Murphy, H. Gall, and P. Devanbu. Don't touch my code!: examining the effects of ownership on software quality. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, ESEC/FSE '11, pages 4–14, New York, NY, USA, 2011. ACM.

[16] D. M. Blei. Probabilistic Topic Models. *Commun. ACM*, 55(4):77–84, Apr. 2012.

[17] L. Breiman. Random forests. *Machine Learning*, 45(1):5–32, Oct 2001.

[18] M. Broy, F. Deissenboeck, and M. Pizka. Demystifying maintainability. In *Proceedings of the 2006 international workshop on Software quality*, WoSQ '06, pages 21–26, New York, NY, USA, 2006. ACM.

[19] F. Budinsky, S. A. Brodsky, and E. Merks. *Eclipse Modeling Framework*. Pearson Education, 2003.

[20] R. P. L. Buse and T. Zimmermann. Information Needs for Software Development Analytics. Technical Report MSR-TR-2011-8, Microsoft Research, Jan. 2011.

[21] R. P. L. Buse and T. Zimmermann. Information needs for software development analytics. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 987–996, June 2012.

[22] G. Canfora and L. Cerulo. Fine grained indexing of software repositories to support impact analysis. In *Proceedings of the 2006 international workshop on Mining software repositories*, pages 105–111, Shanghai, China, 2006. ACM.

[23] G. Canfora, L. Cerulo, and M. Di Penta. Tracking Your Changes: A Language-Independent Approach. *Software, IEEE*, 26(1):50 –57, Feb. 2009.

[24] G. Canfora, L. Cerulo, M. Di Penta, and F. Pacilio. An Exploratory Study of Factors Influencing Change Entropy. In *Program Comprehension (ICPC), 2010 IEEE 18th International Conference on*, pages 134 –143, July 2010.

[25] L. Capretz and F. Ahmed. Making Sense of Software Development and Personality Types. *IT Professional*, 12(1):6 –13, Feb. 2010.

[26] L. F. Capretz. Personality types in software engineering. *International Journal of Human-Computer Studies*, 58(2):207–214, Feb. 2003.

[27] J. M. Chambers. *Graphical methods for data analysis*. Belmont, Calif. : Wadsworth International Group ; Boston : Duxbury Press, 1983.

[28] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer. SMOTE: Synthetic Minority Over-sampling Technique. *arXiv:1106.1813 [cs]*, June 2011. arXiv: 1106.1813.

[29] H. Chen, B. Schatz, T. Ng, J. Martinez, A. Kirchhoff, and C. Lin. A parallel computing approach to creating engineering concept spaces for semantic retrieval: the Illinois Digital Library Initiative project. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 18(8):771–782, Aug. 1996.

[30] S. R. Chidamber and C. Kemerer. A Metric Suite for Object-Oriented Design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

[31] D. R. Cox. The regression analysis of binary sequences (with discussion). *J Roy Stat Soc B*, 20:215–242, 1958.

[32] J. Czerwonka, N. Nagappan, W. Schulte, and B. Murphy. CODEMINE: Building a Software Development Data Analytics Platform at Microsoft. *IEEE Software*, 30(4):64–71, July 2013.

[33] A. D. Da Cunha and D. Greathead. Does personality matter?: an analysis of code-review ability. *Commun. ACM*, 50(5):109–112, May 2007.

[34] J.-m. Dalle, P. A. David, J.-m. Dalle, and P. A. David. *SimCode: Agent-based Simulation Modelling of Open-source Software Development*. 2004.

[35] M. D'Ambros, M. Lanza, and H. Gall. Fractal Figures: Visualizing Development Effort for CVS Entities. In *3rd IEEE International Workshop on Visualizing Software for Understanding and Analysis, 2005. VISSOFT 2005*, pages 1–6, 2005.

[36] G. Daniel, G. Sunyé, A. Benelallam, M. Tisi, Y. Vernageau, A. Gómez, and J. Cabot. NeoEMF: A multi-database model persistence framework for very large models. *Science of Computer Programming*, 149:9–14, Dec. 2017.

[37] S. Datta, R. Sindhgatta, and B. Sengupta. Evolution of developer collaboration on the jazz platform: a study of a large scale agile project. In *Proceedings of the 4th India Software Engineering Conference*, ISEC '11, pages 21–30, New York, NY, USA, 2011. ACM.

[38] S. Demeyer, S. Ducasse, and O. Nierstrasz. Finding refactorings via change metrics. In *Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '00, pages 166–177, New York, NY, USA, 2000. ACM.

[39] M. Dias, A. Bacchelli, G. Gousios, D. Cassou, and S. Ducasse. Untangling Fine-Grained Code Changes. *arXiv:1502.06757 [cs]*, Feb. 2015. arXiv: 1502.06757.

[40] T. A. dos Santos, R. M. de Araujo, and A. M. Magdaleno. Bringing out Collaboration in Software Development Social Networks. In *Proceedings of the 12th International Conference on Product Focused Software Development and Process Improvement*, Profes '11, pages 18–21, New York, NY, USA, 2011. ACM.

[41] P. Drotár and Z. Smékal. Comparison of stability measures for feature selection. In *2015 IEEE 13th International Symposium on Applied Machine Intelligence and Informatics (SAMI)*, pages 71–75, Jan. 2015.

[42] C. Drummond and R. C. Holte. C4.5, class imbalance, and cost sensitivity: Why under-sampling beats over-sampling. pages 1–8, 2003.

[43] E. Duala-Ekoko and M. Robillard. Tracking Code Clones in Evolving Software. In *Software Engineering, 2007. ICSE 2007. 29th International Conference on*, pages 158–167, 2007.

[44] S. Ducasse, N. Anquetil, M. U. Bhatti, A. C. Hora, J. Laval, and T. Girba. MSE and FAMIX 3.0: an Interexchange Format and Source Code Model Family. Nov. 2011.

[45] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen. Boa: a language and infrastructure for analyzing ultra-large-scale software repositories. ICSE '13, pages 422–431, Piscataway, NJ, USA, 2013. IEEE Press.

[46] M. D'Ambros, M. Lanza, and R. Robbes. Evaluating defect prediction approaches: a benchmark and an extensive comparison. *Empirical Software Engineering*, 17(4-5):531–577, Aug. 2012.

[47] J. Ekanayake, J. Tappolet, H. Gall, and A. Bernstein. Time variance and defect prediction in software projects. *Empirical Software Engineering*, 17(4):348–389, 2012.

[48] B. Fluri, M. Wuersch, M. PInzger, and H. Gall. Change Distilling:Tree Differencing for Fine-Grained Source Code Change Extraction. *IEEE Transactions on Software Engineering*, 33(11):725–743, Nov. 2007.

[49] M. Fowler. *Analysis Patterns: Reusable Object Models*. Addison-Wesley, 1997.

[50] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[51] T. Fritz, G. C. Murphy, and E. Hill. Does a programmer's activity indicate knowledge of code? In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC-FSE '07, pages 341–350, New York, NY, USA, 2007. ACM.

[52] T. M. J. Fruchterman and E. M. Reingold. Graph drawing by force-directed placement. *Software: Practice and Experience*, 21(11):1129–1164, Nov. 1991.

[53] D. German. Mining CVS repositories, the softChange experience. *IEE Seminar Digests*, 2004(917):17–21, Jan. 2004.

[54] D. German and A. Hindle. Measuring fine-grained change in software: towards modification-aware change metrics. In *Software Metrics, 2005. 11th IEEE International Symposium*, pages 10 pp.–28, 2005.

[55] D. M. German, P. C. Rigby, and M.-A. Storey. Using evolutionary annotations from change logs to enhance program comprehension. In *Proceedings of the 2006 international workshop on Mining software repositories*, pages 159–162, Shanghai, China, 2006. ACM.

[56] T. Gîrba, J.-M. Favre, and S. Ducasse. Using meta-model transformation to model software evolution. *Electronic Notes in Theoretical Computer Science*, 137(3):57–64, Sept. 2005.

[57] T. Girba, A. Kuhn, M. Seeberger, and S. Ducasse. How developers drive software evolution. In *Eighth International Workshop on Principles of Software Evolution*, pages 113 – 122, Sept. 2005.

[58] M. Godfrey and Q. Tu. Tracking structural evolution using origin analysis. In *Proceedings of the international workshop on Principles of software evolution - IWPSE '02*, page 117, Orlando, Florida, 2002.

[59] M. Godfrey and L. Zou. Using origin analysis to detect merging and splitting of source code entities. *Software Engineering, IEEE Transactions on*, 31(2):166–181, 2005.

[60] M. Goeminne, M. Claes, and T. Mens. A historical dataset for the gnome ecosystem. MSR '13, pages 225–228, Piscataway, NJ, USA, 2013. IEEE Press.

[61] M. Goeminne and T. Mens. A framework for analysing and visualising open source software ecosystems. IWPSE-EVOL '10, pages 42–47, New York, NY, USA, 2010. ACM.

[62] G. Gousios, E. Kalliamvakou, and D. Spinellis. Measuring developer contribution from software repository data. In *Proceedings of the 2008 international working conference on Mining software repositories*, pages 129–132, Leipzig, Germany, 2008. ACM.

[63] G. Gousios and D. Spinellis. A platform for software engineering research. pages 31–40, Vancouver, BC, Canada, May 2009.

[64] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The weka data mining software: an update. *ACM SIGKDD Explorations Newsletter*, 11(1):10–18, 2009.

[65] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell. A Systematic Literature Review on Fault Prediction Performance in Software Engineering. *IEEE Transactions on Software Engineering*, 38(6):1276–1304, Nov. 2012.

[66] A. Hassan. The road ahead for mining software repositories. In *ICSE Workshops MSR '07. Fourth International Workshop on Mining Software Repositories*, pages 48–57, 2008.

[67] H. Hata, O. Mizuno, and T. Kikuno. Historage: Fine-grained Version Control System for Java. In *Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th Annual ERCIM Workshop on Software Evolution*, IWPSE-EVOL '11, pages 96–100, New York, NY, USA, 2011. ACM.

[68] L. Hattori and M. Lanza. On the nature of commits. pages 63–71, 2008.

[69] Q. He, B. Shen, and Y. Chen. Software Defect Prediction Using Semi-Supervised Learning with Change Burst Information. In *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*, volume 1, pages 113–122, June 2016.

[70] Z. He, F. Shu, Y. Yang, M. Li, and Q. Wang. An investigation on the feasibility of cross-project defect prediction. *Automated Software Engineering*, 19:167–199, 2012.

[71] F. Heider. *The Psychology of Interpersonal Relations*. Psychology Press, 1958.

[72] H. Hemmati, S. Nadi, O. Baysal, O. Kononenko, W. Wang, R. Holmes, and M. W. Godfrey. The MSR cookbook: mining a decade of research. MSR '13, pages 343–352, Piscataway, NJ, USA, 2013. IEEE Press.

[73] B. Henderson-Sellers. *Object-Oriented Metrics*. Prentice Hall, 1996.

[74] S. Herbold. Training Data Selection for Cross-project Defect Prediction. In *Proceedings of the 9th International Conference on Predictive Models in Software Engineering*, PROMISE '13, pages 6:1–6:10, New York, NY, USA, 2013. ACM.

[75] S. Herbold, A. Trautsch, and J. Grabowski. Global vs. local models for cross-project defect prediction. *Empirical Software Engineering*, 22(4):1866–1902, Aug. 2017.

[76] K. Herzig and A. Zeller. The impact of tangled code changes. MSR '13, page 121–130, Piscataway, NJ, USA, 2013. IEEE Press.

[77] A. Hindle. *Evidence-based Software Process Recovery*. PhD Thesis, University of Waterloo, 2010.

[78] A. Hindle. Green mining: A methodology of relating software change to power consumption. pages 78–87, 2012.

[79] A. Hindle, D. M. German, and R. Holt. What do large commits tell us?: a taxonomical study of large commits. In *Proceedings of the 2008 international working conference on Mining software repositories*, pages 99–108, Leipzig, Germany, 2008. ACM.

[80] D. Honsel, V. Honsel, M. Welter, S. Waack, and J. Grabowski. Monitoring Software Quality by Means of Simulation Methods. In *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '16, pages 11:1–11:6, New York, NY, USA, 2016. ACM.

[81] V. Honsel, S. Herbold, and J. Grabowski. Hidden Markov Models for the Prediction of Developer Involvement Dynamics and Workload. In *Proceedings of the The 12th International Conference on Predictive Models and Data Analytics in Software Engineering*, PROMISE 2016, pages 8:1–8:10, New York, NY, USA, 2016. ACM.

[82] V. Honsel, D. Honsel, J. Grabowski, and S. Waack. Developer Oriented and Quality Assurance Based Simulation of Software Processes. In *Proceedings of the Seminar Series on Advanced Techniques and Tools for Software Evolution (SATToSE) 2015*, July 2015.

[83] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones. In *Proceedings of the 29th international conference on Software Engineering*, pages 96–105. IEEE Computer Society, 2007.

[84] T. Jiang, L. Tan, and S. Kim. Personalized defect prediction. In *2013 IEEE/ACM 28th International Conference on Automated Software Engineering (ASE)*, pages 279–289, Nov. 2013.

[85] Y. Jiang, B. Cukic, and Y. Ma. Techniques for evaluating fault prediction models. *Empirical Softw. Engg.*, 13(5):561–595, 2008.

[86] A. Kalousis, J. Prados, and M. Hilario. Stability of Feature Selection Algorithms: A Study on High-dimensional Spaces. *Knowl. Inf. Syst.*, 12(1):95–116, May 2007.

[87] J. Karn and T. Cowling. A follow up study of the effect of personality on the performance of software engineering teams. In *Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*, ISESE '06, pages 232–241, New York, NY, USA, 2006. ACM.

[88] I. Keivanloo, C. Forbes, A. Hmood, M. Erfani, C. Neal, G. Peristerakis, and J. Rilling. A linked data platform for mining software repositories. pages 32–35, 2012.

[89] S. Kelly and J.-P. Tolvanen. *Domain-Specific Modeling: Enabling Full Code Generation*. Aug. 2007.

[90] T. Khoshgoftaar, E. Allen, N. Goel, A. Nandi, and J. McMullan. Detection of software modules with high debug code churn in a very large legacy system. In *Proceedings of the Seventh International Symposium on Software Reliability Engineering*, pages 364–371, Oct. 1996.

[91] T. Khoshgoftaar and R. Szabo. Improving code churn predictions during the system test and maintenance phases. In *Proceedings of the International Conference on Software Maintenance*, pages 58–67, Sept. 1994.

[92] C. Kiefer, A. Bernstein, and J. Tappolet. Mining software repositories with iSPARQL and a software evolution ontology. In *ICSE Workshops MSR '07. Fourth International Workshop on Mining Software Repositories*, page 10, 2007.

[93] M. Kim and D. Notkin. Discovering and Representing Systematic Code Changes. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 309–319, Washington, DC, USA, 2009. IEEE Computer Society.

[94] M. Kim, V. Sazawal, and D. Notkin. An empirical study of code clone genealogies. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering - ESEC/FSE-13*, page 187, Lisbon, Portugal, 2005.

[95] S. Kim, E. Whitehead, and Y. Zhang. Classifying software changes: Clean or buggy? *IEEE Transactions on Software Engineering*, 34(2):181–196, Apr. 2008.

[96] S. Kim, E. J. Whitehead, and J. Bevan. Analysis of signature change patterns. In *Proceedings of the 2005 international workshop on Mining software repositories*, pages 1–5, St. Louis, Missouri, 2005. ACM.

[97] S. Kim, T. Zimmermann, K. Pan, and E. Whitehead. Automatic Identification of Bug-Introducing Changes. In *Automated Software Engineering, 2006. ASE '06. 21st IEEE/ACM International Conference on*, pages 81–90, 2006.

[98] D. S. Kolovos, R. F. Paige, and F. A. C. Polack. The epsilon transformation language. In A. Vallecillo, J. Gray, and A. Pierantonio, editors, *Theory and Practice of Model Transformations*, number 5063 in Lecture Notes in Computer Science, pages 46–60. Springer Berlin Heidelberg, Jan. 2008.

[99] J. Krinke. Identifying similar code with program dependence graphs. In *Proceedings of the Eighth Working Conference on Reverse Engineering*, pages 301–309, 2001.

[100] S. Krishnan, C. Strasburg, R. R. Lutz, and K. Goševa-Popstojanova. Are change metrics good predictors for an evolving software product line? In *Proceedings of the 7th International Conference on Predictive Models in Software Engineering*, Promise '11, pages 7:1–7:10, New York, NY, USA, 2011. ACM.

[101] T. K. Landauer, P. W. Foltz, and D. Laham. An introduction to latent semantic analysis. *Discourse Processes*, 25:259–284, 1998.

[102] M. Lanza. The Evolution Matrix: Recovering Software Evolution Using Software Visualization Techniques. In *Proceedings of the 4th International Workshop on Principles of Software Evolution*, IWPSE '01, pages 37–42, New York, NY, USA, 2001. ACM.

[103] M. Lanza and R. Marinescu. *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*. Springer Publishing Company, Incorporated, 1st edition, 2010.

[104] M. Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060 – 1076, Sept. 1980.

[105] M. M. Lehman. Laws of Software Evolution Revisited. In *Proceedings of the 5th European Workshop on Software Process Technology*, EWSPT '96, pages 108–124, London, UK, UK, 1996. Springer-Verlag.

[106] M. M. Lehman. Approach to a theory of software evolution. In *Eighth International Workshop on Principles of Software Evolution*, pages 135–, Sept. 2005.

[107] M. M. Lehman, G. Kahen, and J. F. Ramil. Behavioural modelling of long-lived evolution processes: some issues and an example. *Journal of Software Maintenance*, 14(5):335–351, Sept. 2002.

[108] D. D. Lewis and J. Catlett. Heterogenous Uncertainty Sampling for Supervised Learning. In *Proceedings of the Eleventh International Conference on International Conference on Machine Learning*, ICML'94, pages 148–156, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[109] S. A. Licorish and S. G. MacDonell. Personality Profiles of Global Software Developers. In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*, EASE '14, pages 45:1–45:10, New York, NY, USA, 2014. ACM.

[110] G. S. Linoff and M. J. A. Berry. *Data Mining Techniques: For Marketing, Sales, and Customer Relationship Management*. Wiley Publishing, 3rd edition, 2011.

[111] E. Linstead, P. Rigor, S. Bajracharya, C. Lopes, and P. Baldi. Mining Eclipse Developer Contributions via Author-Topic Models. In *Mining Software Repositories, 2007. ICSE Workshops MSR '07. Fourth International Workshop on*, page 30, 2007.

[112] M. Lumpe, R. Vasa, T. Menzies, R. Rush, and B. Turhan. Learning better inspection optimization policies. *International Journal of Software Engineering and Knowledge Engineering*, 22(05):621–644, Aug. 2012.

[113] L. Madeyski and M. Kawalerowicz. Continuous defect prediction: The idea and a related dataset. In *Proceedings of the 14th International Conference on Mining Software Repositories*, MSR '17, pages 515–518, Piscataway, NJ, USA, 2017. IEEE Press.

[114] P. Makedonski and J. Grabowski. Weighted Multi-Factor Multi-Layer Identification of Potential Causes for Events of Interest in Software Repositories. In *Proceedings of the Seminar Series on Advanced Techniques and Tools for Software Evolution (SATToSE) 2015*, July 2015.

[115] P. Makedonski, J. Grabowski, and F. Philipp. Quantifying the evolution of TTCN-3 as a language. *International Journal on Software Tools for Technology Transfer*, 16(3):227–246, July 2013.

[116] P. Makedonski, F. Sudau, and J. Grabowski. Towards a model-based software mining infrastructure. *SIGSOFT Softw. Eng. Notes*, 40(1):1–8, Feb. 2015.

[117] P. Marinescu, P. Hosek, and C. Cadar. Covrig: A Framework for the Analysis of Code, Test, and Coverage Evolution in Real Software. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 93–104, New York, NY, USA, 2014. ACM.

[118] S. Matsumoto, Y. Kamei, A. Monden, K.-i. Matsumoto, and M. Nakamura. An Analysis of Developer Metrics for Fault Prediction. In *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*, PROMISE '10, pages 18:1–18:9, New York, NY, USA, 2010. ACM.

[119] T. J. McCabe. A Complexity Measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, 1976.

[120] T. Menzies, C. Bird, T. Zimmermann, W. Schulte, and E. Kocaganeli. The inductive software engineering manifesto: principles for industrial data mining. MALETS '11, page 19–26, New York, NY, USA, 2011. ACM.

[121] T. Menzies, B. Caglayan, E. Kocaguneli, J. Krall, F. Peters, and B. Turhan. The promise repository of empirical software engineering data, June 2012.

[122] T. Menzies, J. Greenwald, and A. Frank. Data mining static code attributes to learn defect predictors. *IEEE Trans. Softw. Eng.*, 33(1):2–13, Jan. 2007.

[123] A. Miranskyy, B. Caglayan, A. Bener, and E. Cialini. Effect of Temporal Collaboration Network, Maintenance Activity, and Experience on Defect Exposure. In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '14, pages 27:1–27:8, New York, NY, USA, 2014. ACM.

[124] T. M. Mitchell. *Machine Learning*. McGraw-Hill, Inc., New York, NY, USA, 1 edition, 1997.

[125] L. Moonen. Generating robust parsers using island grammars. In *Eighth Working Conference on Reverse Engineering, 2001. Proceedings*, pages 13–22, 2001.

[126] R. Moser, W. Pedrycz, and G. Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 181–190, New York, NY, USA, 2008. ACM.

[127] N. Nagappan and T. Ball. Use of relative code churn measures to predict system defect density. In *27th International Conference on Software Engineering, 2005. ICSE 2005*, pages 284–292. IEEE, May 2005.

[128] N. Nagappan, T. Ball, and A. Zeller. Mining metrics to predict component failures. In *Proceedings of the ACM/IEEE 28th International Conference on Software engineering*, ICSE '06, pages 452–461, New York, NY, USA, 2006. ACM.

[129] N. Nagappan, A. Zeller, T. Zimmermann, K. Herzig, and B. Murphy. Change Bursts as Defect Predictors. In *2010 IEEE 21st International Symposium on Software Reliability Engineering*, pages 309–318, Nov. 2010.

[130] M. Niazi and A. Hussain. Agent-based computing from multi-agent systems to agent-based models: a visual survey. *Scientometrics*, 89(2):479, Nov. 2011.

[131] O. Nierstrasz. Agile software assessment with Moose. *SIGSOFT Softw. Eng. Notes*, 37(3):1–5, May 2012.

[132] Object Management Group (OMG). Uml 2.4.1 superstructure specification, 2011.

[133] Object Management Group OMG. Object Constraint Language, Version 2.3.1. OMG Document Number: formal/2012-05-09, Standard document URL: `http://www.omg.org/spec/OCL/2.3.1/`, May 2012.

[134] Object Management Group (OMG). Meta object facility core, v2.4.2, 2014.

[135] Object Management Group OMG. XML Metadata Interchange (XMI), Version 2.4.2. OMG Document Number: formal/2014-04-06, Standard document URL: `http://www.omg.org/spec/XMI/2.5.1/`, April 2014.

[136] Object Management Group (OMG). Mof query / view / transformation, 2016.

[137] M. Ohba and K. Gondow. Toward mining "concept keywords" from identifiers in large software projects. In *Proceedings of the 2005 international workshop on Mining software repositories*, pages 1–5, St. Louis, Missouri, 2005. ACM.

[138] M. Ohira, R. Yokomori, M. Sakai, K. Matsumoto, K. Inoue, and K. Torii. Empirical project monitor: a tool for mining multiple project data. *IEE Seminar Digests*, 2004(917):42–46, Jan. 2004.

[139] T. Ostrand, E. Weyuker, and R. Bell. Predicting the location and number of faults in large software systems. *IEEE Transactions on Software Engineering*, 31(4):340–355, 2005.

[140] S. J. Pan and Q. Yang. A Survey on Transfer Learning. *IEEE Transactions on Knowledge and Data Engineering*, 22(10):1345–1359, Oct. 2010.

[141] D. Posnett, R. D'Souza, P. Devanbu, and V. Filkov. Dual Ecological Measures of Focus in Software Development. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 452–461, Piscataway, NJ, USA, 2013. IEEE Press.

[142] L. Prechelt and A. Pepper. Why Software Repositories Are Not Used for Defect-insertion Circumstance Analysis More Often: A Case Study. *Inf. Softw. Technol.*, 56(10):1377–1389, Oct. 2014.

[143] J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[144] F. Rahman and P. Devanbu. Ownership, experience and defects: a fine-grained study of authorship. ICSE '11, page 491–500, New York, NY, USA, 2011. ACM.

[145] B. Raskutti, H. L. Ferrá, and A. Kowalczyk. Second Order Features for Maximising Text Classification Performance. In *Proceedings of the 12th European Conference on Machine Learning*, EMCL '01, pages 419–430, London, UK, UK, 2001. Springer-Verlag.

[146] M. Riaz, E. Mendes, and E. Tempero. A systematic review of software maintainability prediction and metrics. In *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*, ESEM '09, pages 367–377, Washington, DC, USA, 2009. IEEE Computer Society.

[147] R. Robbes. Mining a change-based software repository. In *Fourth International Workshop on Mining Software Repositories, 2007. ICSE Workshops MSR '07*, pages 15–15, 2007.

[148] R. Robbes and M. Lanza. A change-based approach to software evolution. *Electronic Notes in Theoretical Computer Science*, 166:93–109, Jan. 2007.

[149] M. Robillard, W. Coelho, and G. Murphy. How effective developers investigate source code: an exploratory study. *IEEE Transactions on Software Engineering*, 30(12):889–903, 2004.

[150] W. N. Robinson and T. Deng. Data Mining Behavioral Transitions in Open Source Repositories. In *2015 48th Hawaii International Conference on System Sciences*, pages 5280–5289, Jan. 2015.

[151] G. Robles. Replicating MSR: a study of the potential replicability of papers published in the mining software repositories proceedings. In *2010 7th IEEE*

Working Conference on Mining Software Repositories (MSR), pages 171–180, 2010.

[152] G. Robles and J. M. Gonzalez-Barahona. Developer identification methods for integrated data from various sources. In *Proceedings of the 2005 international workshop on Mining software repositories*, pages 1–5, St. Louis, Missouri, 2005. ACM.

[153] G. Robles, S. Koch, J. M. González-Barahona, and J. Carlos. Remote analysis and measurement of libre software systems by means of the CVSAnalY tool. In *In Proceedings of the 2nd ICSE Workshop on Remote Analysis and Measurement of Software Systems (RAMSS*, pages 51–55, 2004.

[154] T. Roehm and W. Maalej. Automatically detecting developer activities and problems in software development work. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 1261–1264, Piscataway, NJ, USA, 2012. IEEE Press.

[155] C. K. Roy and J. R. Cordy. A Survey on Software Clone Detection Research. *School of Computing TR 2007-541, Queen's University*, 115, 2007.

[156] S. J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 2 edition, 2003.

[157] C. Sadowski, C. Lewis, Z. Lin, X. Zhu, and E. J. Whitehead,Jr. An empirical analysis of the FixCache algorithm. In *Proceeding of the 8th working conference on Mining software repositories*, MSR '11, pages 219–222, New York, NY, USA, 2011. ACM.

[158] N. Salman and A. Doğru. Design effort estimation using complexity metrics. *J. Integr. Des. Process Sci.*, 8(3):83–88, Aug. 2004.

[159] M. Scheidgen. Reference representation techniques for large models. BigMDE '13, pages 5:1–5:9, New York, NY, USA, 2013. ACM.

[160] N. Schneidewind and M. Hinchey. A Complexity Reliability Model. In *Software Reliability Engineering, 2009. ISSRE '09. 20th International Symposium on*, pages 1–10, 2009.

[161] D. Schuler and T. Zimmermann. Mining usage expertise from version archives. MSR '08, page 121–124, New York, NY, USA, 2008. ACM.

[162] M. Shepperd, Q. Song, Z. Sun, and C. Mair. Data quality: Some comments on the nasa software defect datasets. *IEEE Transactions on Software Engineering*, 39(9):1208–1215, Sept 2013.

[163] E. Shihab, A. E. Hassan, B. Adams, and Z. M. Jiang. An industrial study on the risk of software changes. FSE '12, pages 62:1–62:11, New York, NY, USA, 2012. ACM.

[164] J. Siegel. Object management group model driven architecture (mda) mda guide rev. 2.0. 2014.

[165] J. Śliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? In *Proceedings of the 2005 international workshop on Mining software repositories*, pages 1–5, St. Louis, Missouri, 2005. ACM.

[166] N. Smith, A. Capiluppi, and J. Fernández-Ramil. Users and developers: an agent-based simulation of open source software evolution. In *Proceedings of the 2006 international conference on Software Process Simulation and Modeling*, SPW/ProSim'06, pages 286–293, Berlin, Heidelberg, 2006. Springer-Verlag.

[167] N. Smith, A. Capiluppi, and J. F. Ramil. Agent-based Simulation of Open Source Evolution. *Software Process: Improvement and Practice*, July 2006.

[168] J. Spacco, J. Strecker, D. Hovemeyer, and W. Pugh. Software repository mining with marmoset: an automated programming project snapshot and testing system. In *MSR '05 Proceedings of the 2005 international workshop on Mining software repositories*, pages 1–5, St. Louis, Missouri, 2005. ACM.

[169] D. F. Specht. A general regression neural network. *IEEE Transactions on Neural Networks*, 2(6):568–576, Nov 1991.

[170] B. Stopford and S. Counsell. A framework for the simulation of structural software evolution. *ACM Trans. Model. Comput. Simul.*, 18(4):17:1–17:36, Sept. 2008.

[171] F. Sudau, T. Friede, J. Grabowski, J. Koschack, P. Makedonski, and W. Himmel. Sources of Information and Behavioral Patterns in Health Online Forums. *Journal of Medical Internet Research*, 2014. JMIR - Publications.

[172] A. Sureka, A. Goyal, and A. Rastogi. Using social network analysis for mining collaboration data in a defect tracking system for risk and vulnerability analysis. In *Proceedings of the 4th India Software Engineering Conference*, ISEC '11, pages 195–204, New York, NY, USA, 2011. ACM.

[173] M. Tan, L. Tan, S. Dara, and C. Mayeux. Online Defect Prediction for Imbalanced Data. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 99–108, May 2015.

[174] F. Trautsch, S. Herbold, P. Makedonski, and J. Grabowski. Addressing Problems with External Validity of Repository Mining Studies Through a Smart Data Platform. In *Proceedings of the 13th International Conference on Mining Software Repositories*, pages 97–108, Austin, Texas, 2016. ACM.

[175] F. Trautsch, S. Herbold, P. Makedonski, and J. Grabowski. Addressing problems with replicability and validity of repository mining studies through a smart data platform. *Empirical Software Engineering*, Aug. 2017.

[176] B. Turhan, T. Menzies, A. B. Bener, and J. Di Stefano. On the Relative Value of Cross-company and Within-company Data for Defect Prediction. *Empirical Softw. Engg.*, 14(5):540–578, Oct. 2009.

[177] P. Turney. Technical note: Bias and the quantification of stability. *Machine Learning*, 20(1-2):23–33, July 1995.

[178] T. Van Gestel, J. A. K. Suykens, B. Baesens, S. Viaene, J. Vanthienen, G. Dedene, B. De Moor, and J. Vandewalle. Benchmarking least squares support vector machine classifiers. *Mach. Learn.*, 54(1):5–32, Jan. 2004.

[179] D. Varona, L. F. Capretz, Y. Piñero, and A. Raza. Evolution of software engineers' personality profile. *SIGSOFT Softw. Eng. Notes*, 37(1):1–5, Jan. 2012.

[180] G. Venolia. Textual Allusions to Artifacts in Software-Related Repositories. In *Proceedings of the 2006 international workshop on Mining software repositories*, pages 151–154, Shanghai, China, 2006. ACM.

[181] G. von Krogh, S. Spaeth, and K. R. Lakhani. Community, joining, and specialization in open source software innovation: a case study. *Research Policy*, 32(7):1217–1241, July 2003.

[182] A. Wald. Sequential Tests of Statistical Hypotheses. *The Annals of Mathematical Statistics*, 16(2):117–186, June 1945.

[183] P. Weissgerber and S. Diehl. Identifying Refactorings from Source-Code Changes. In *21st IEEE/ACM International Conference on Automated Software Engineering, 2006. ASE '06*, pages 231–240. IEEE, Sept. 2006.

[184] E. J. Weyuker, T. J. Ostrand, and R. M. Bell. Using developer information as a factor for fault prediction. page 8. IEEE Computer Society, 2007.

[185] T. Wickenberg and P. Davidsson. On Multi Agent Based Simulation of Software Development Processes. In J. Simão Sichman, F. Bousquet, and P. Davidsson, editors, *Multi-Agent-Based Simulation II*, volume 2581 of *Lecture Notes in Computer Science*, pages 72–77. Springer Berlin / Heidelberg, 2003.

[186] I. S. Wiese, F. R. Côgo, R. Ré, I. Steinmacher, and M. A. Gerosa. Social Metrics Included in Prediction Models on Software Engineering: A Mapping Study. In *Proceedings of the 10th International Conference on Predictive Models in Software Engineering*, PROMISE '14, pages 72–81, New York, NY, USA, 2014. ACM.

[187] C. Williams and J. Spacco. SZZ revisited: verifying when changes induce fixes. In *Proceedings of the 2008 workshop on Defects in large software systems*, DEFECTS '08, pages 32–36, New York, NY, USA, 2008. ACM. ACM ID: 1390826.

[188] T. Wolf, A. Schroter, D. Damian, L. Panjer, and T. Nguyen. Mining Task-Based Social Networks to Explore Collaboration in Software Teams. *IEEE Software*, 26(1):58–66, Jan. 2009.

[189] X. Xia, D. Lo, X. Wang, and X. Yang. Collective Personalized Change Classification With Multiobjective Search. *IEEE Transactions on Reliability*, 65(4):1810–1829, Dec. 2016.

[190] J. Xu, Y. Gao, S. Christley, and G. Madey. A Topological Analysis of the Open Souce Software Development Community. In *Proceedings of the 38th Annual Hawaii International Conference on System Sciences, 2005. HICSS '05*, pages 198a–198a, Jan. 2005.

[191] N. Xu. *An exploratory study of open source software based on public project archives*. masters, Concordia University, 2003.

[192] A. Ying and M. Robillard. The Influence of the Task on Programmer Behaviour. In *2011 IEEE 19th International Conference on Program Comprehension (ICPC)*, pages 31–40, 2011.

[193] L. Yu and S. Ramaswamy. Mining CVS repositories to understand open-source project developer roles. In *ICSE Workshops MSR '07. Fourth International Workshop on Mining Software Repositories*, page 8, 2007.

[194] A. Zaidman, B. Van Rompaey, S. Demeyer, and A. van Deursen. Mining Software Repositories to Study Co-Evolution of Production #x00026; Test Code. In *2008 1st International Conference on Software Testing, Verification, and Validation*, pages 220–229, Apr. 2008.

[195] D. Zhang, S. Han, Y. Dang, J.-G. Lou, H. Zhang, and T. Xie. Software Analytics in Practice. *IEEE Software*, 30(5):30–37, Sept. 2013.

[196] H. Zhang, B. Kitchenham, and D. Pfahl. Software process simulation modeling: an extended systematic review. In *Proceedings of the 2010 international*

conference on New modeling concepts for today's software processes: software process, ICSP'10, pages 309–320, Berlin, Heidelberg, 2010. Springer-Verlag.

[197] T. Zhu, Y. Wu, X. Peng, Z. Xing, and W. Zhao. Monitoring Software Quality Evolution by Analyzing Deviation Trends of Modularity Views. In *2011 18th Working Conference on Reverse Engineering (WCRE)*, pages 229 –238, Oct. 2011.

[198] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, and B. Murphy. Cross-project Defect Prediction: A Large Scale Experiment on Data vs. Domain vs. Process. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, pages 91–100, New York, NY, USA, 2009. ACM.

[199] T. Zimmermann, N. Nagappan, P. J. Guo, and B. Murphy. Characterizing and predicting which bugs get reopened. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 1074–1083, Piscataway, NJ, USA, 2012. IEEE Press.

[200] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl. Mining version histories to guide software changes. *IEEE Transactions on Software Engineering*, 31(6):429–445, June 2005.

[201] I. Čavrak, M. Orlić, and I. Crnković. Collaboration patterns in distributed software development projects. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 1235–1244, Piscataway, NJ, USA, 2012. IEEE Press.

A. Appendix

The appendix complements the thesis with additional information that was considered too detailed, complicated, or technical for the main text. The contents can be helpful for reproducing and extending the work described in this thesis.

A.1. Additional Views on the FAMIX Meta-model

For the sake of completeness, the additional views on the reconstructed *FAMIX* meta-model, including the concepts related to inheritance, invocation, and variables shown in Figure A.1, Figure A.2, and Figure A.3.

A.2. Model-based Software Mining Infrastructure

The instantiation of the model-based software mining infrastructure is published online as a collection of open source projects[43]. A dedicated project[44] provides an overview of general technical aspects related to the instantiation and further technical information related to this thesis, including technical specifications and representations of the meta-models, mappings, transformations, and metrics. Where applicable, the individual projects include further technical and usage-related information specific the corresponding project.

A.3. Enlarged Figures

Enlarged versions of selected figures are included for reference.

[43]See `https://github.com/DECENTSoftwareAssessment`
[44]See `https://github.com/DECENTSoftwareAssessment/DECENT.Documentation`

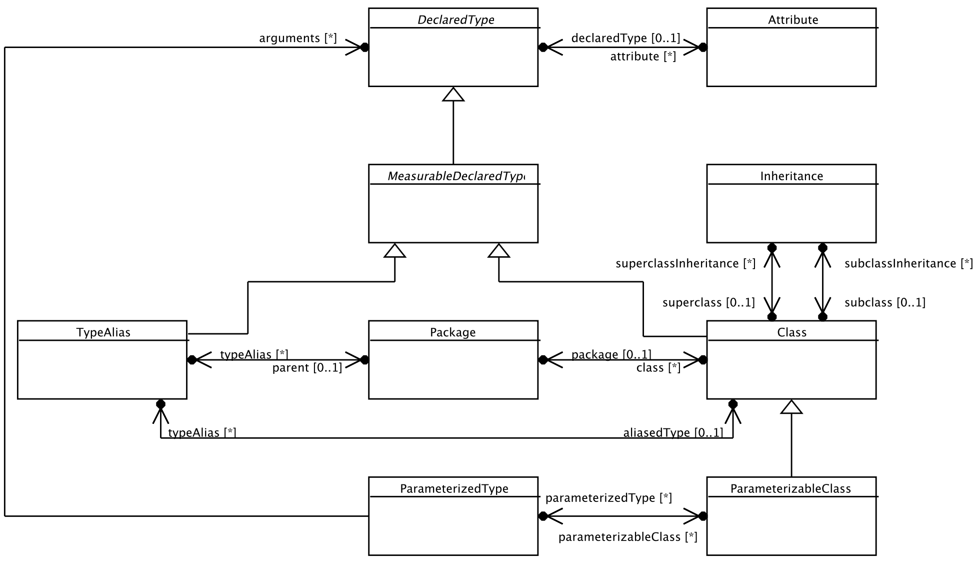


Figure A.1.: *FAMIX* meta-model based on the structure of data extracted with *InFamix* (inheritance-part)

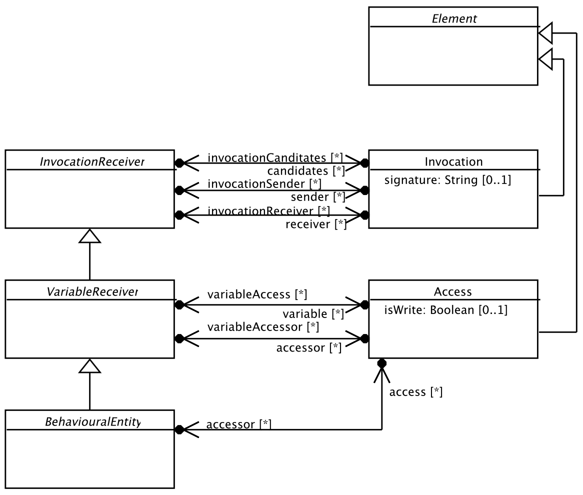


Figure A.2.: *FAMIX* meta-model based on the structure of data extracted with *InFamix* (invocation-part)

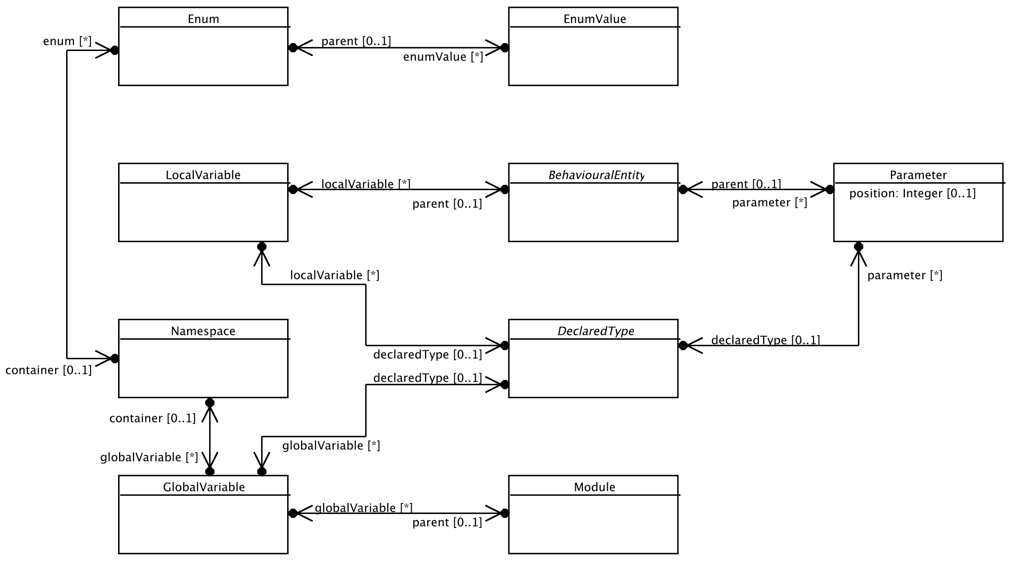
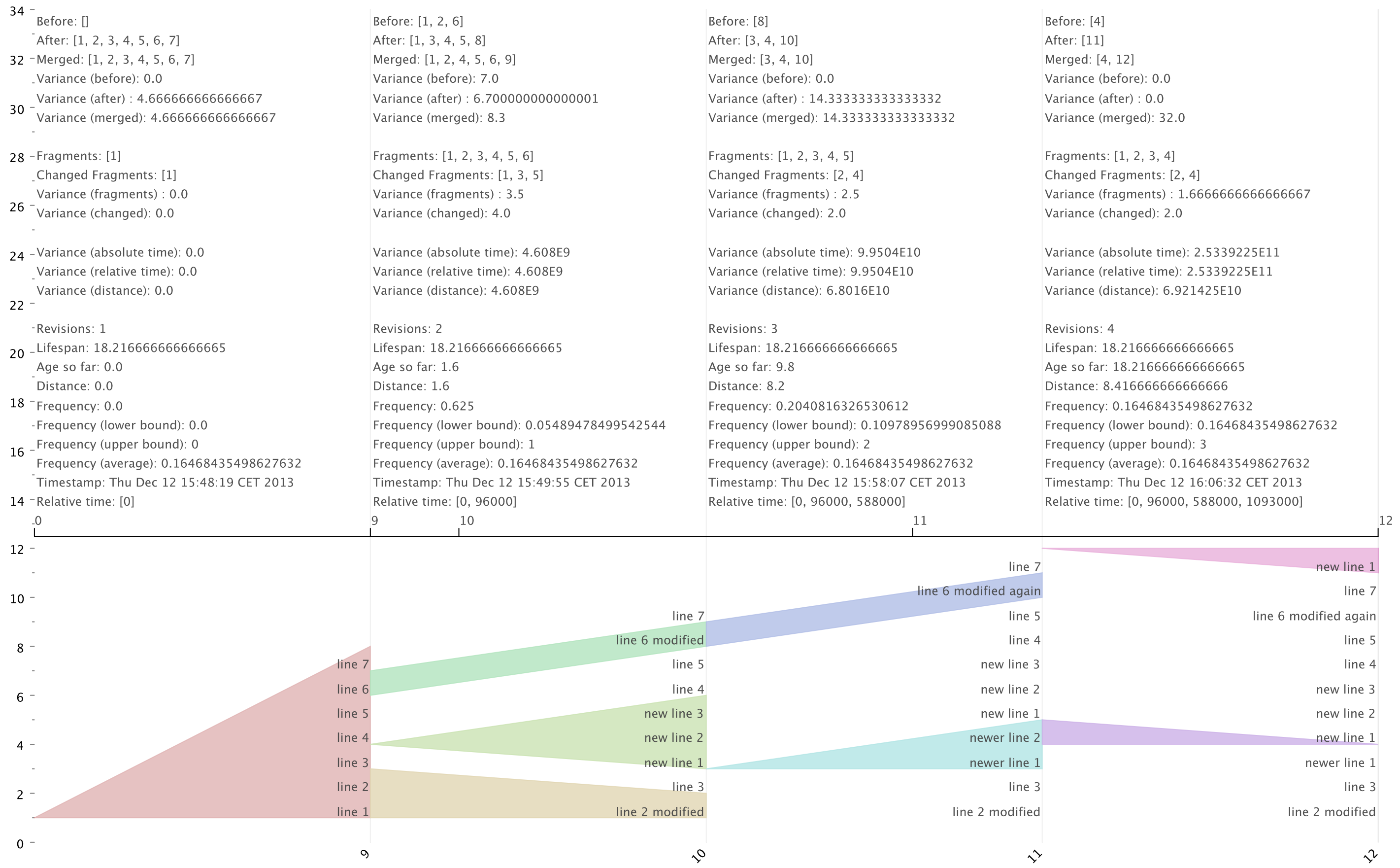


Figure A.3.: *FAMIX* meta-model based on the structure of data extracted with *InFamix* (variable-part)



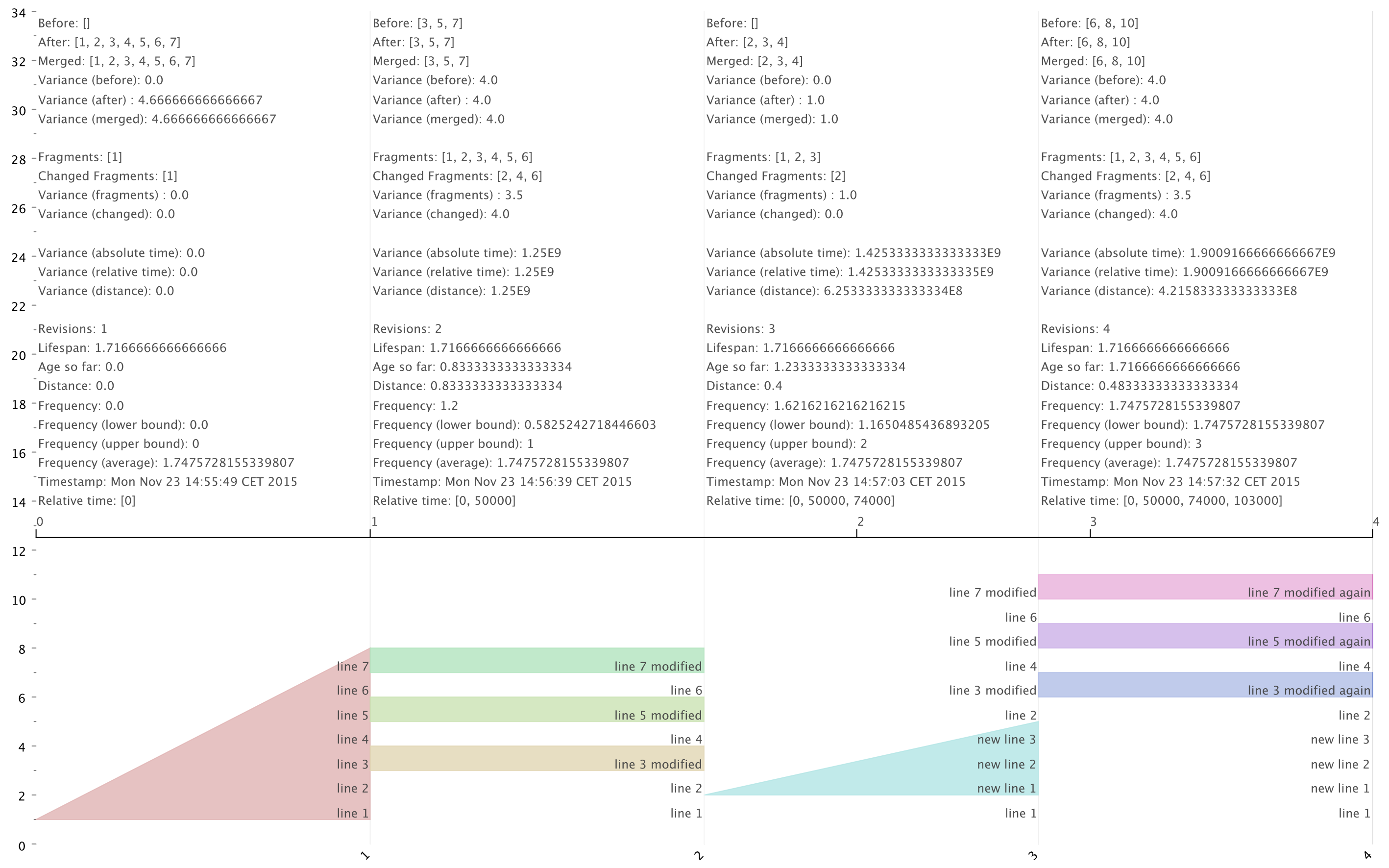
```
Before: []
After: [1, 2, 3, 4, 5, 6, 7]
Merged: [1, 2, 3, 4, 5, 6, 7]
Variance (before): 0.0
Variance (after) : 4.666666666666667
Variance (merged): 4.666666666666667

Fragments: [1]
Changed Fragments: [1]
Variance (fragments) : 0.0
Variance (changed): 0.0

Variance (absolute time): 0.0
Variance (relative time): 0.0
Variance (distance): 0.0

Revisions: 1
Lifespan: 18.216666666666665
Age so far: 0.0
Distance: 0.0
Frequency: 0.0
Frequency (lower bound): 0.0
Frequency (upper bound): 0
Frequency (average): 0.16468435498627632
Timestamp: Thu Dec 12 15:48:19 CET 2013
Relative time: [0]
```

```
Before: [1, 2, 6]
After: [1, 3, 4, 5, 8]
Merged: [1, 2, 4, 5, 6, 9]
Variance (before): 7.0
Variance (after) : 6.700000000000001
Variance (merged): 8.3

Fragments: [1, 2, 3, 4, 5, 6]
Changed Fragments: [1, 3, 5]
Variance (fragments) : 3.5
Variance (changed): 4.0

Variance (absolute time): 4.608E9
Variance (relative time): 4.608E9
Variance (distance): 4.608E9

Revisions: 2
Lifespan: 18.216666666666665
Age so far: 1.6
Distance: 1.6
Frequency: 0.625
Frequency (lower bound): 0.05489478499542544
Frequency (upper bound): 1
Frequency (average): 0.16468435498627632
Timestamp: Thu Dec 12 15:49:55 CET 2013
Relative time: [0, 96000]
```

```
Before: [8]
After: [3, 4, 10]
Merged: [3, 4, 10]
Variance (before): 0.0
Variance (after) : 14.333333333333332
Variance (merged): 14.333333333333332

Fragments: [1, 2, 3, 4, 5]
Changed Fragments: [2, 4]
Variance (fragments) : 2.5
Variance (changed): 2.0

Variance (absolute time): 9.9504E10
Variance (relative time): 9.9504E10
Variance (distance): 6.8016E10

Revisions: 3
Lifespan: 18.216666666666665
Age so far: 9.8
Distance: 8.2
Frequency: 0.2040816326530612
Frequency (lower bound): 0.10978956999085088
Frequency (upper bound): 2
Frequency (average): 0.16468435498627632
Timestamp: Thu Dec 12 15:58:07 CET 2013
Relative time: [0, 96000, 588000]
```

```
Before: [4]
After: [11]
Merged: [4, 12]
Variance (before): 0.0
Variance (after) : 0.0
Variance (merged): 32.0

Fragments: [1, 2, 3, 4]
Changed Fragments: [2, 4]
Variance (fragments) : 1.6666666666666667
Variance (changed): 2.0

Variance (absolute time): 2.5339225E11
Variance (relative time): 2.5339225E11
Variance (distance): 6.921425E10

Revisions: 4
Lifespan: 18.216666666666665
Age so far: 18.216666666666665
Distance: 8.416666666666666
Frequency: 0.16468435498627632
Frequency (lower bound): 0.16468435498627632
Frequency (upper bound): 3
Frequency (average): 0.16468435498627632
Timestamp: Thu Dec 12 16:06:32 CET 2013
Relative time: [0, 96000, 588000, 1093000]
```

Figure A.4.: Spatial characteristics example (large, original in Figure 4.4)



```
Before: []
After: [1, 2, 3, 4, 5, 6, 7]
Merged: [1, 2, 3, 4, 5, 6, 7]
Variance (before): 0.0
Variance (after) : 4.666666666666667
Variance (merged): 4.666666666666667

Fragments: [1]
Changed Fragments: [1]
Variance (fragments) : 0.0
Variance (changed): 0.0

Variance (absolute time): 0.0
Variance (relative time): 0.0
Variance (distance): 0.0

Revisions: 1
Lifespan: 1.7166666666666666
Age so far: 0.0
Distance: 0.0
Frequency: 0.0
Frequency (lower bound): 0.0
Frequency (upper bound): 0
Frequency (average): 1.7475728155339807
Timestamp: Mon Nov 23 14:55:49 CET 2015
Relative time: [0]
```

```
Before: [3, 5, 7]
After: [3, 5, 7]
Merged: [3, 5, 7]
Variance (before): 4.0
Variance (after) : 4.0
Variance (merged): 4.0

Fragments: [1, 2, 3, 4, 5, 6]
Changed Fragments: [2, 4, 6]
Variance (fragments) : 3.5
Variance (changed): 4.0

Variance (absolute time): 1.25E9
Variance (relative time): 1.25E9
Variance (distance): 1.25E9

Revisions: 2
Lifespan: 1.7166666666666666
Age so far: 0.8333333333333334
Distance: 0.8333333333333334
Frequency: 1.2
Frequency (lower bound): 0.5825242718446603
Frequency (upper bound): 1
Frequency (average): 1.7475728155339807
Timestamp: Mon Nov 23 14:56:39 CET 2015
Relative time: [0, 50000]
```

```
Before: []
After: [2, 3, 4]
Merged: [2, 3, 4]
Variance (before): 0.0
Variance (after) : 1.0
Variance (merged): 1.0

Fragments: [1, 2, 3]
Changed Fragments: [2]
Variance (fragments) : 1.0
Variance (changed): 0.0

Variance (absolute time): 1.4253333333333333E9
Variance (relative time): 1.4253333333333335E9
Variance (distance): 6.253333333333334E8

Revisions: 3
Lifespan: 1.7166666666666666
Age so far: 1.2333333333333334
Distance: 0.4
Frequency: 1.6216216216216215
Frequency (lower bound): 1.1650485436893205
Frequency (upper bound): 2
Frequency (average): 1.7475728155339807
Timestamp: Mon Nov 23 14:57:03 CET 2015
Relative time: [0, 50000, 74000]
```

```
Before: [6, 8, 10]
After: [6, 8, 10]
Merged: [6, 8, 10]
Variance (before): 4.0
Variance (after) : 4.0
Variance (merged): 4.0

Fragments: [1, 2, 3, 4, 5, 6]
Changed Fragments: [2, 4, 6]
Variance (fragments) : 3.5
Variance (changed): 4.0

Variance (absolute time): 1.9009166666666667E9
Variance (relative time): 1.9009166666666667E9
Variance (distance): 4.215833333333333E8

Revisions: 4
Lifespan: 1.7166666666666666
Age so far: 1.7166666666666666
Distance: 0.48333333333333334
Frequency: 1.7475728155339807
Frequency (lower bound): 1.7475728155339807
Frequency (upper bound): 3
Frequency (average): 1.7475728155339807
Timestamp: Mon Nov 23 14:57:32 CET 2015
Relative time: [0, 50000, 74000, 103000]
```

Figure A.5.: Recurring changes example (large, original in Figure 4.5)

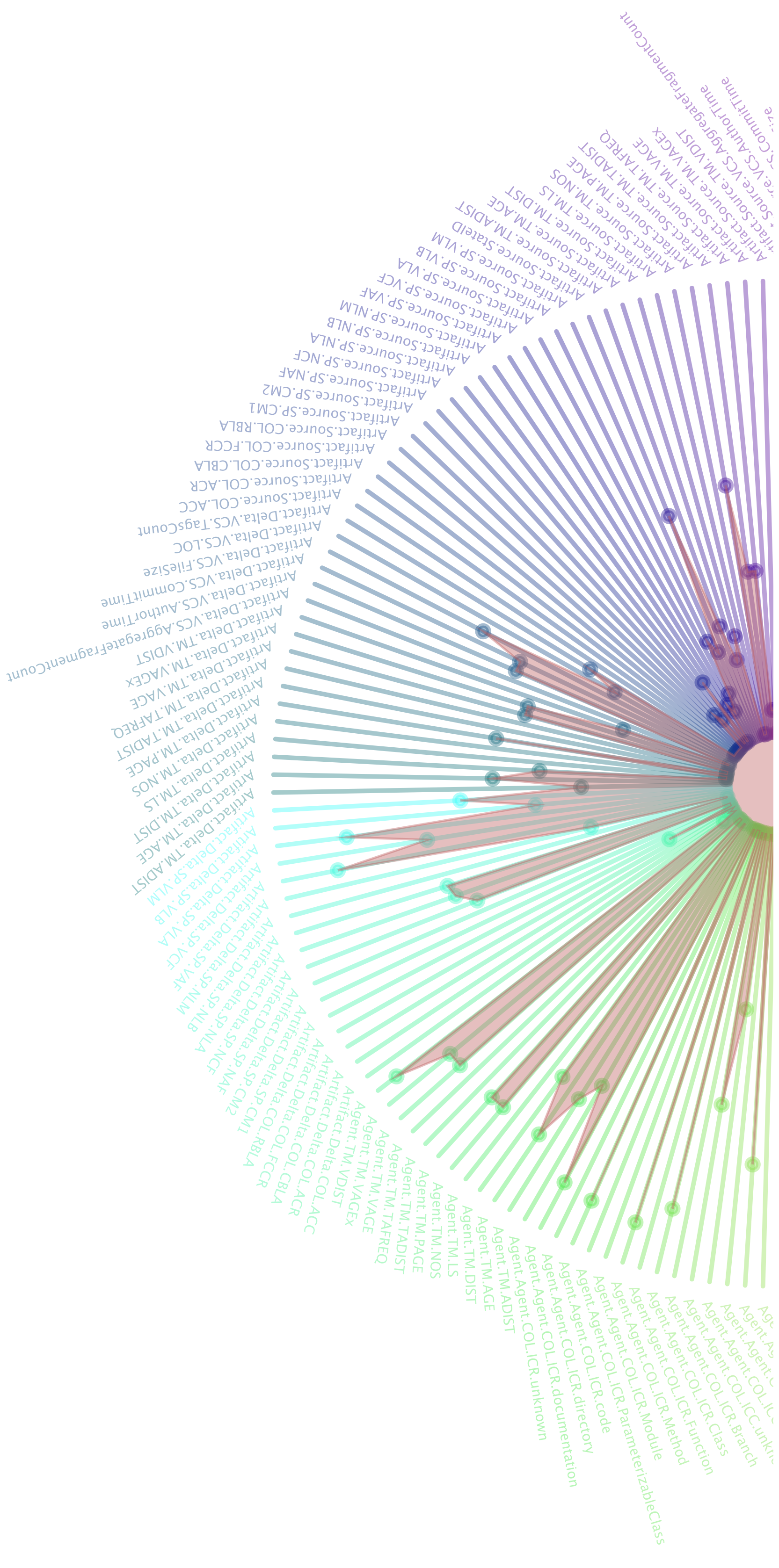


Figure A.6.: Characteristics ranking for developer *A* in log4j (larger, left half, original in Figure 4.6)

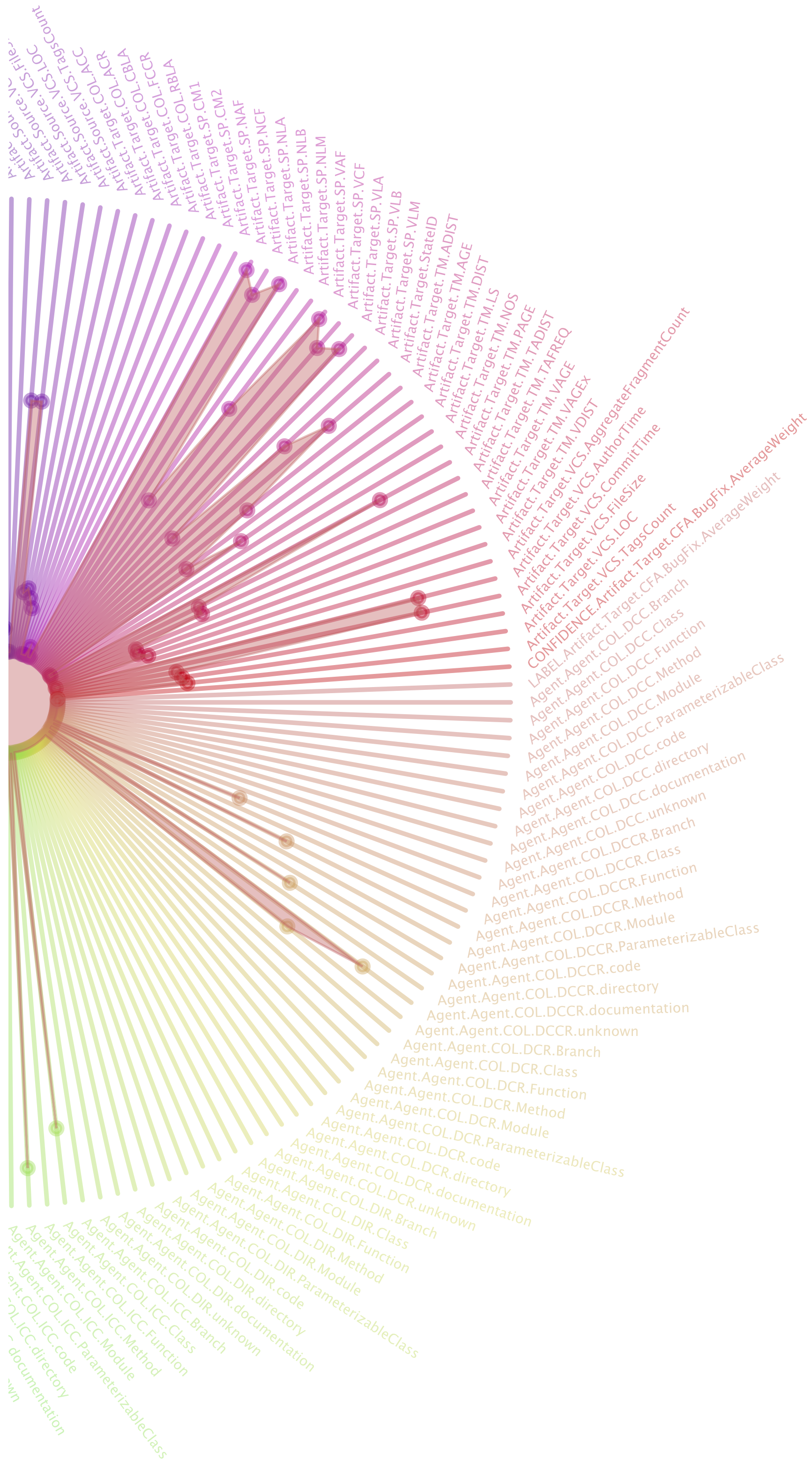


Figure A.7.: Characteristics ranking for developer *A* in log4j (larger, right half, original in Figure 4.6)

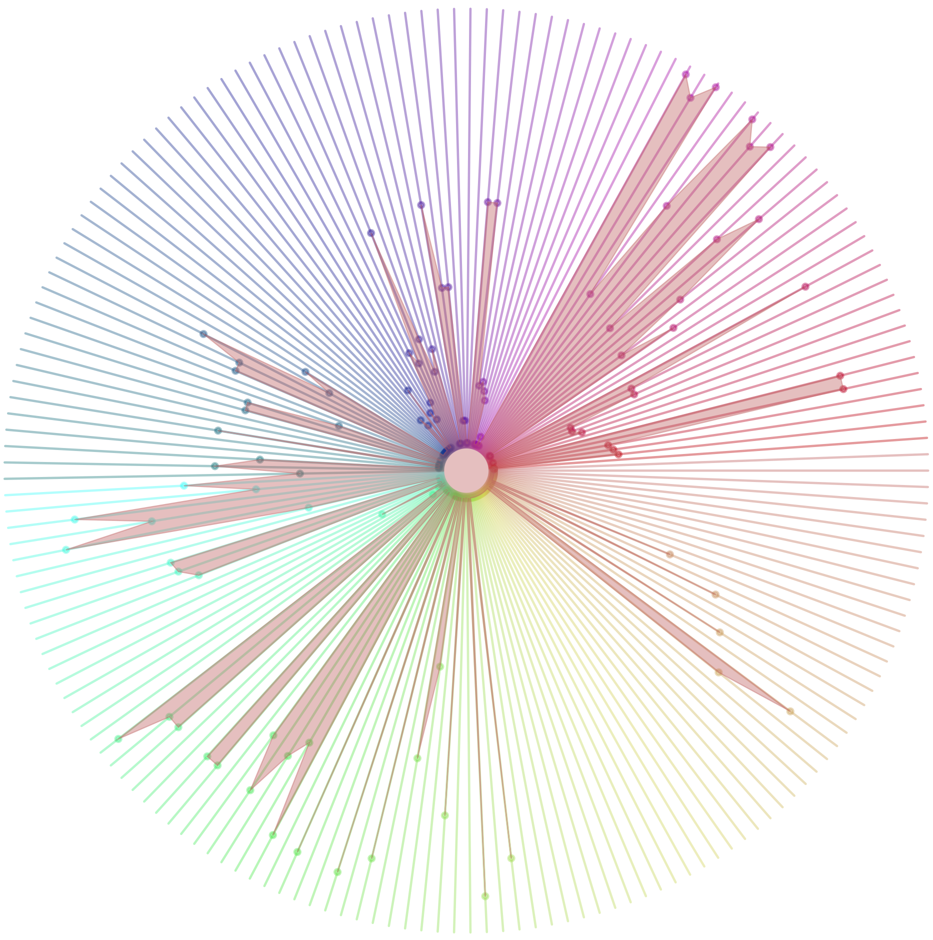


Figure A.8.: Characteristics ranking for developer *A* in log4j (large, original in Figure 4.7a)

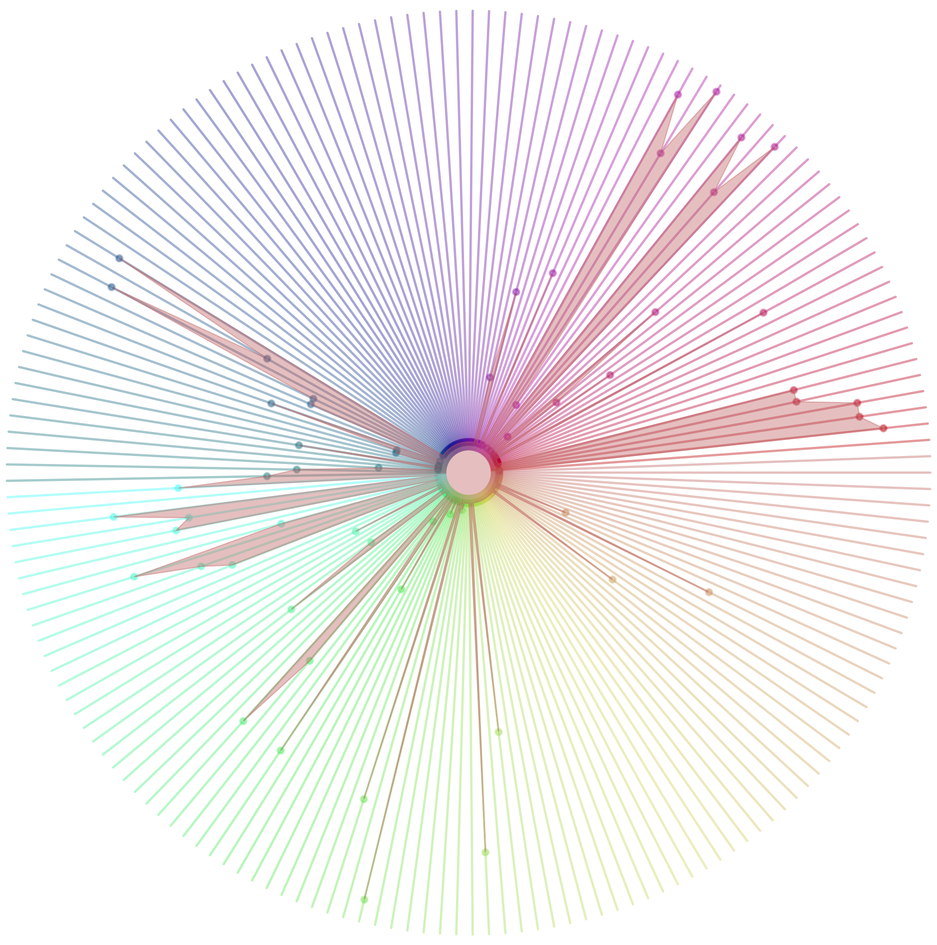


Figure A.9.: Characteristics ranking for developer *A* in log4j (first half, large, original in Figure 4.7b)

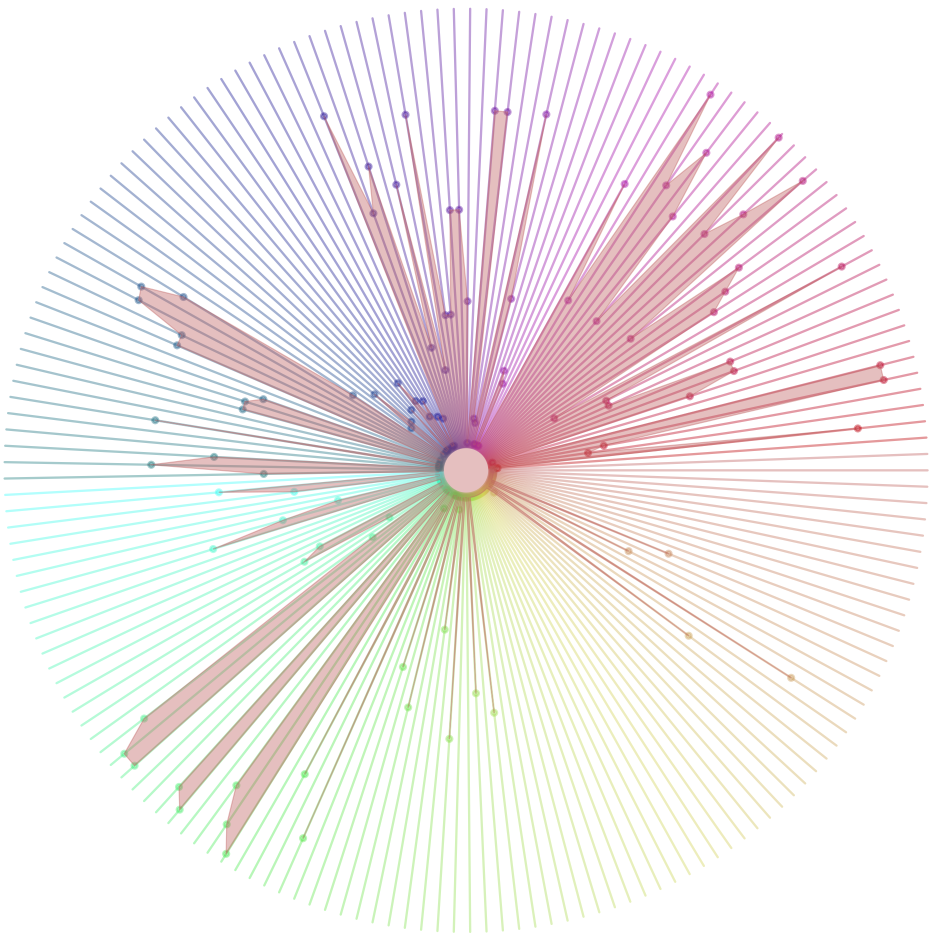


Figure A.10.: Characteristics ranking for developer *B* in log4j (large, original in Figure 4.7c)

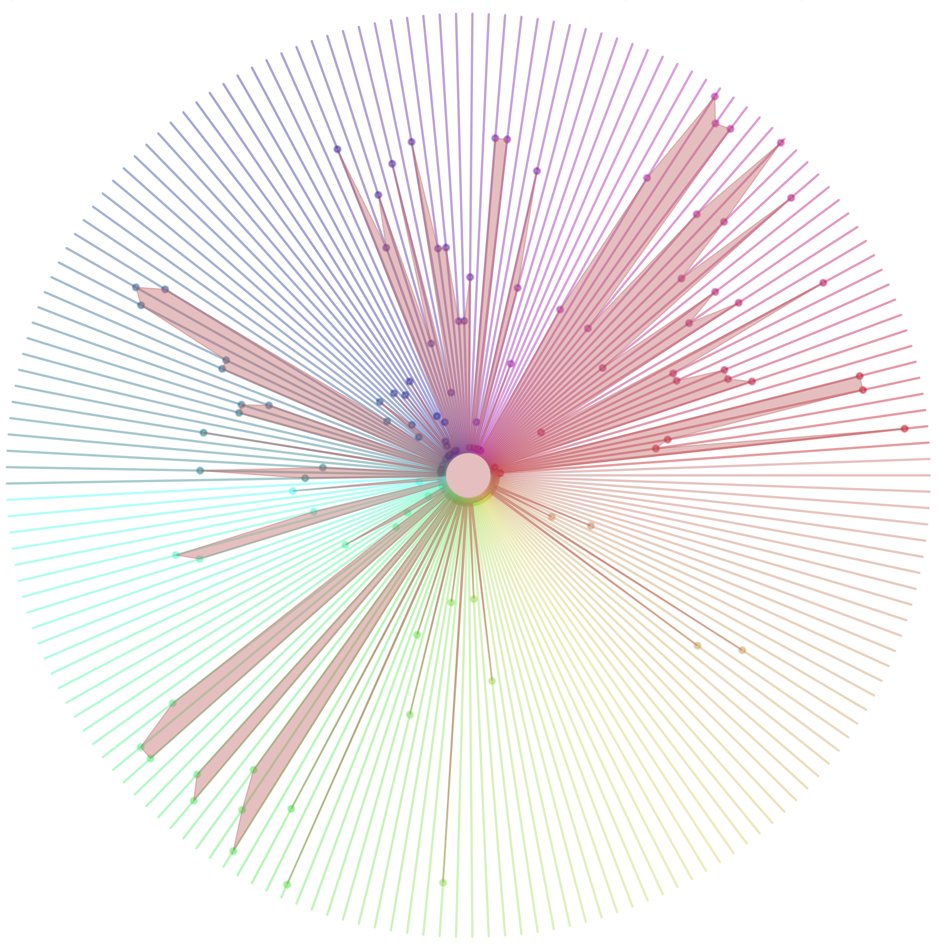


Figure A.11.: Characteristics ranking for developer *B* in log4j (first half, large, original in Figure 4.7d)

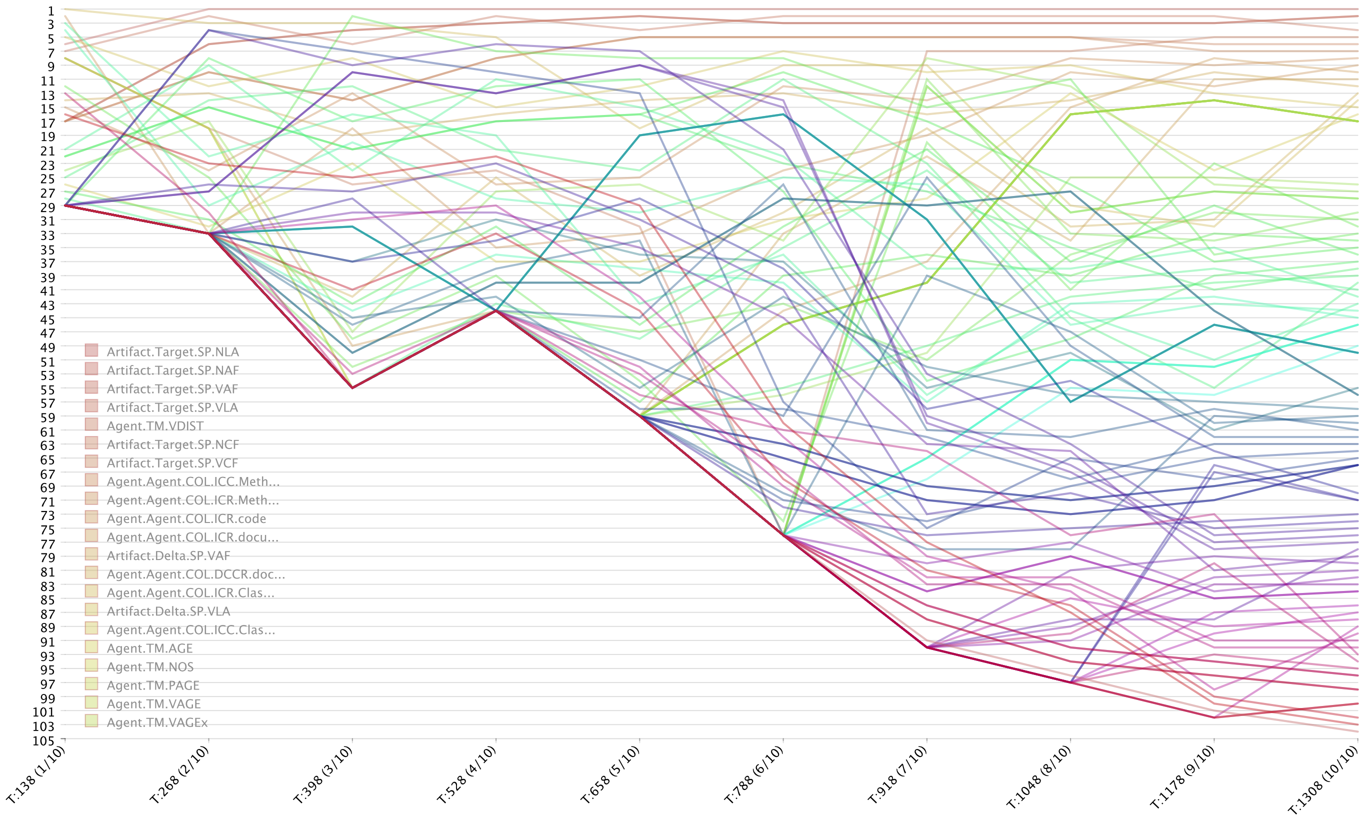


Figure A.12.: Characteristics ranking for developer *A* in log4j over time (large, original in Figure 4.8)

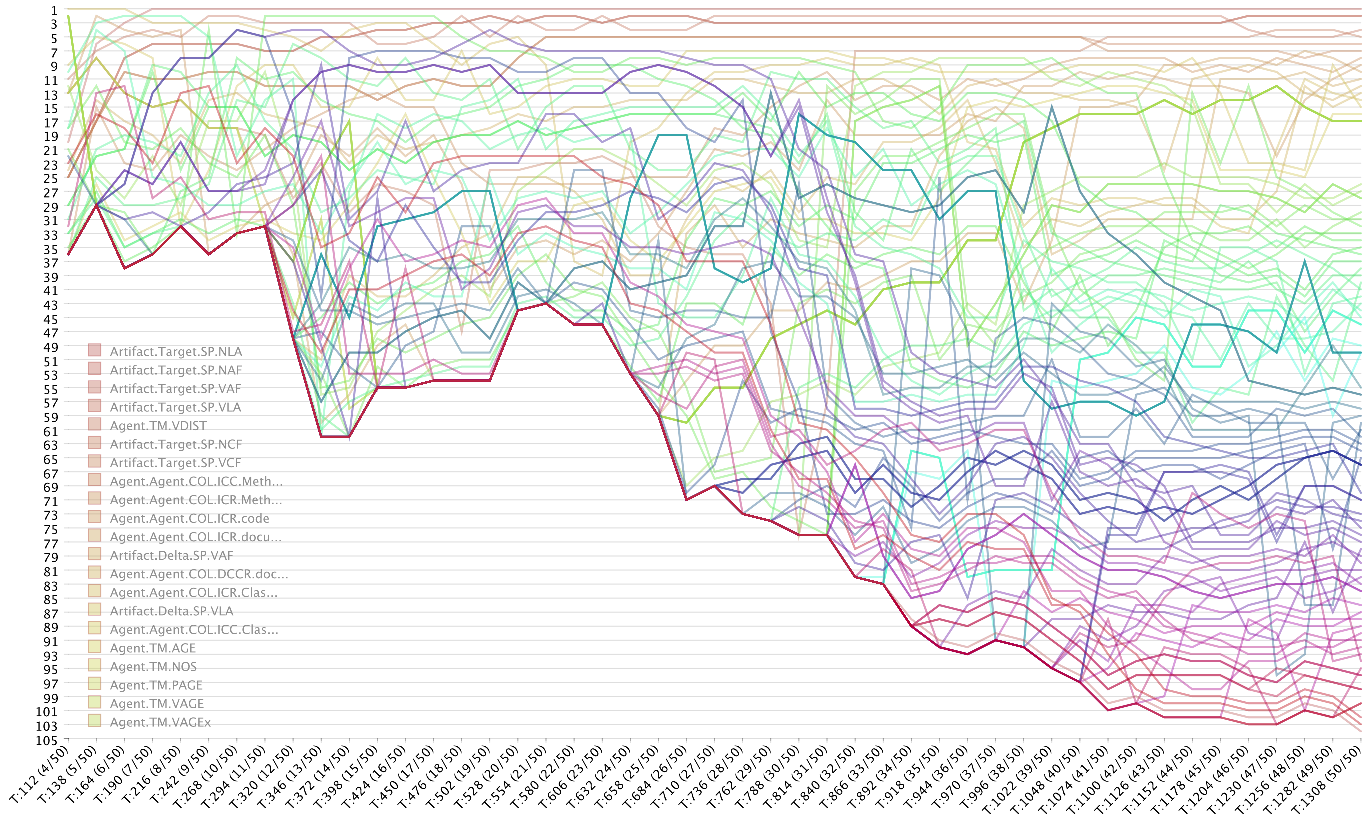


Figure A.13.: Detailed characteristics ranking over time for developer *A* in log4j (large, original in Figure 4.9)

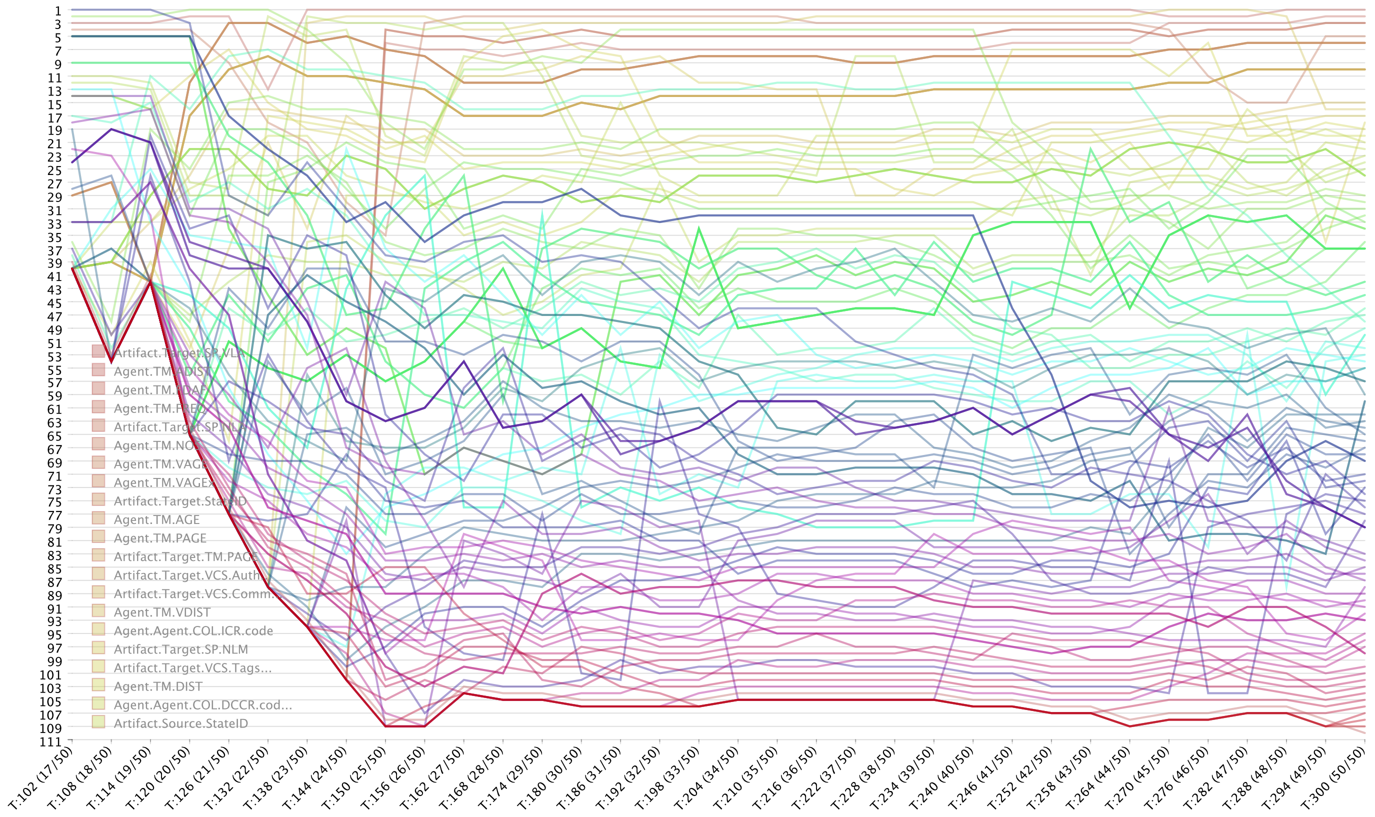


Figure A.14.: Detailed characteristics ranking over time for developer *B* in log4j (large, original in Figure 4.10b)

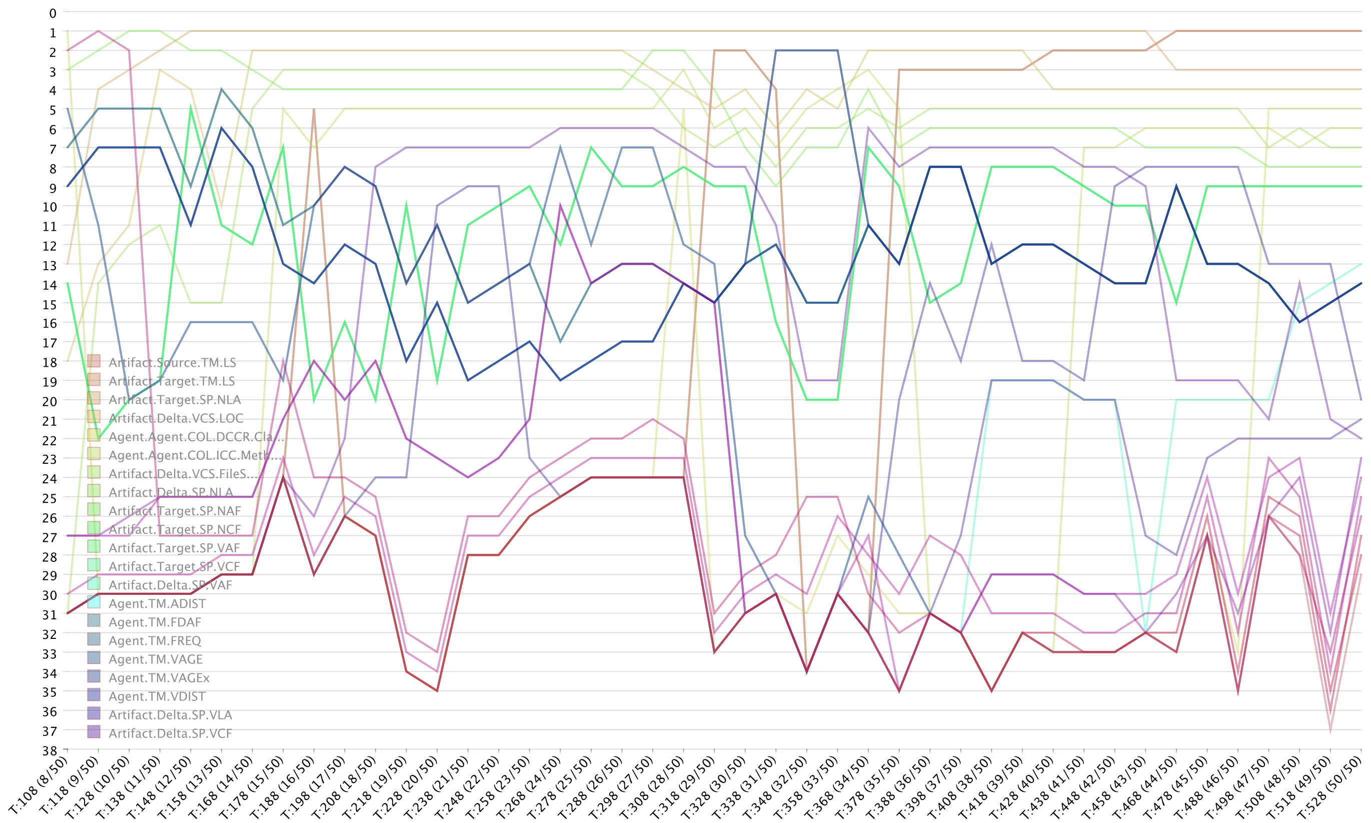


Figure A.15.: Detailed characteristics ranking over time for developer *C* in log4j (large, original in Figure 4.10c)

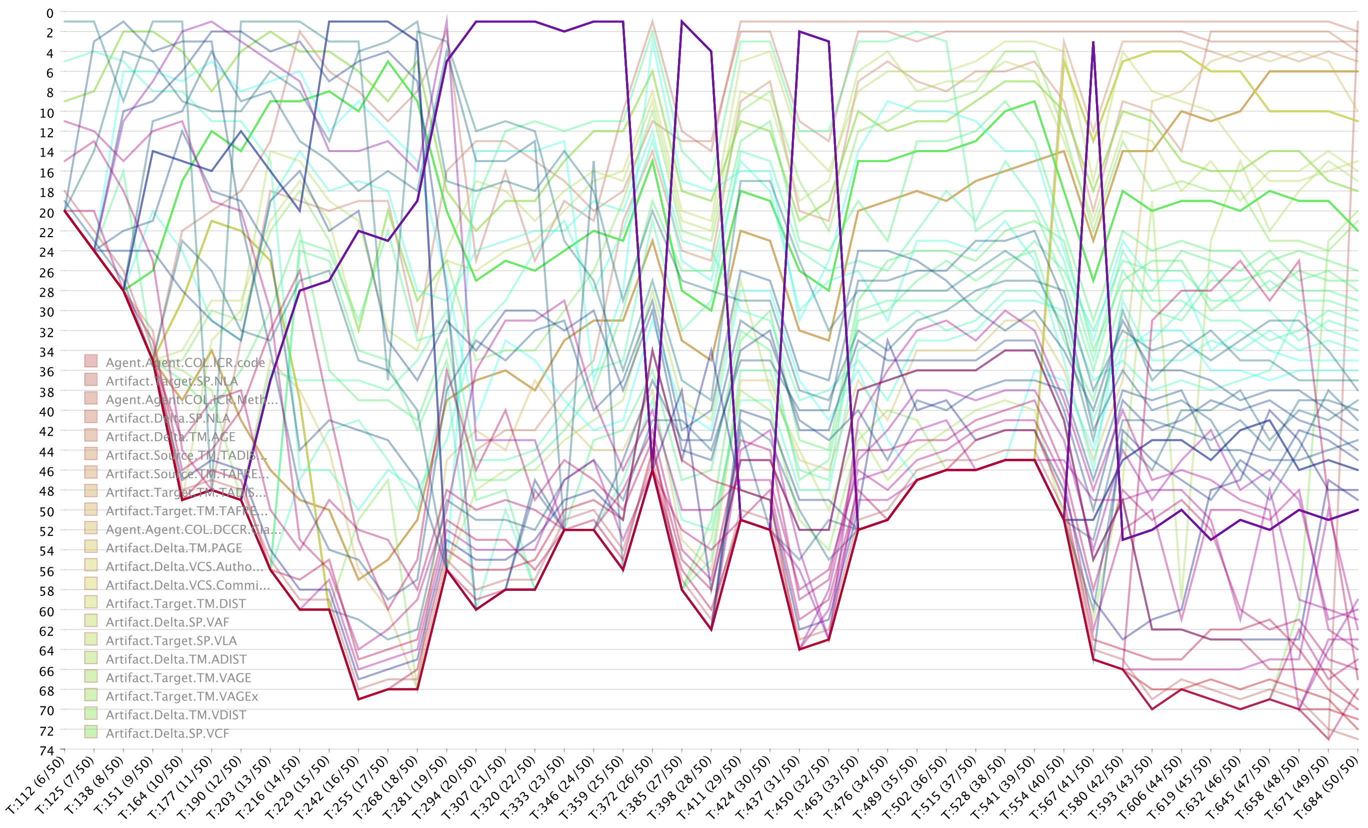


Figure A.16.: Detailed characteristics ranking over time for developer *D* in log4j (large, original in Figure 4.10d)

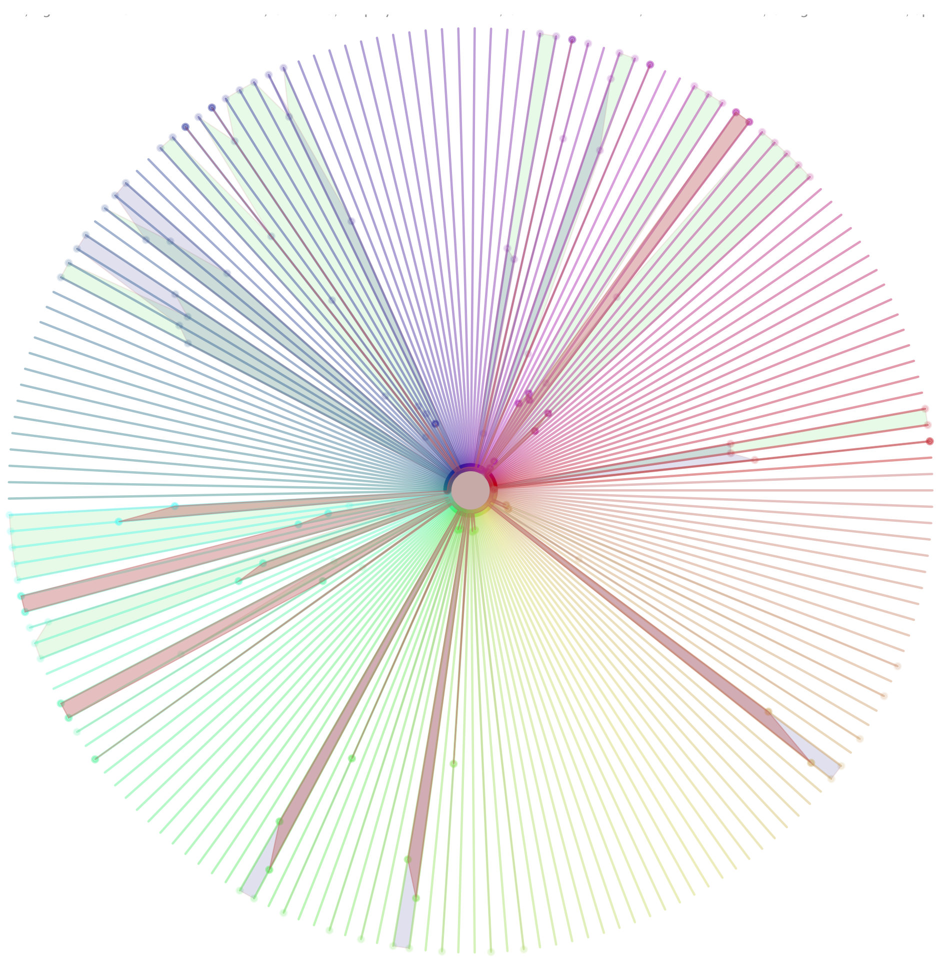


Figure A.17.: Defining characteristics for developer *A* in log4j across three clusters (cluster 1, large, original in Figure 4.18a)

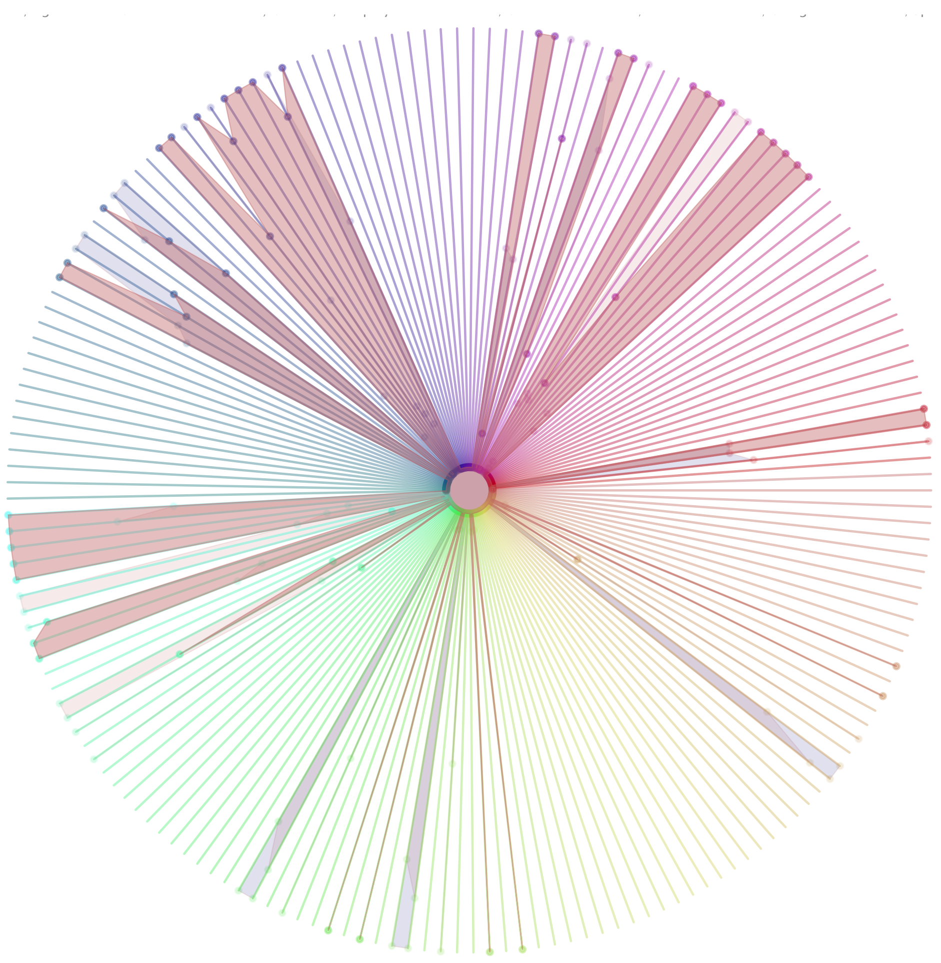


Figure A.18.: Defining characteristics for developer *A* in log4j across three clusters (cluster 2, large, original in Figure 4.18b)

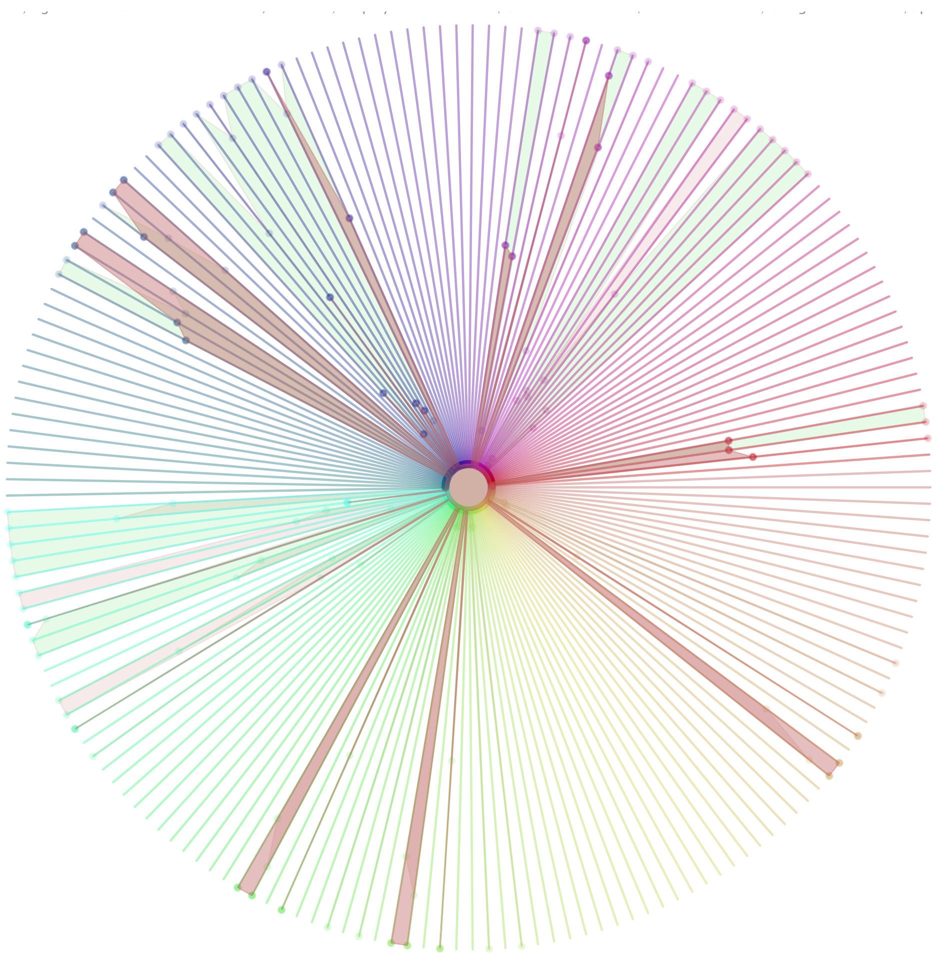


Figure A.19.: Defining characteristics for developer *A* in log4j across three clusters (cluster 3, large, original in Figure 4.18c)

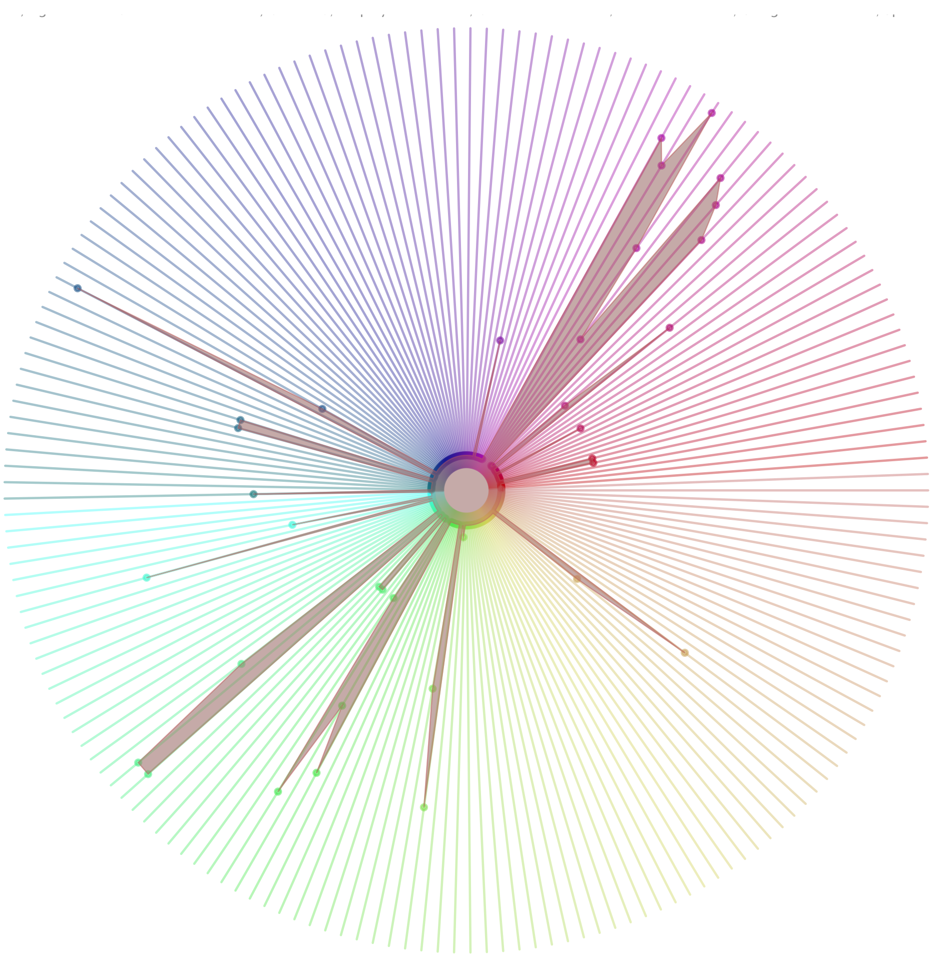


Figure A.20.: Ranking of characteristics for developer *A* in log4j across three clusters (cluster 1, large, original in Figure 4.19a)

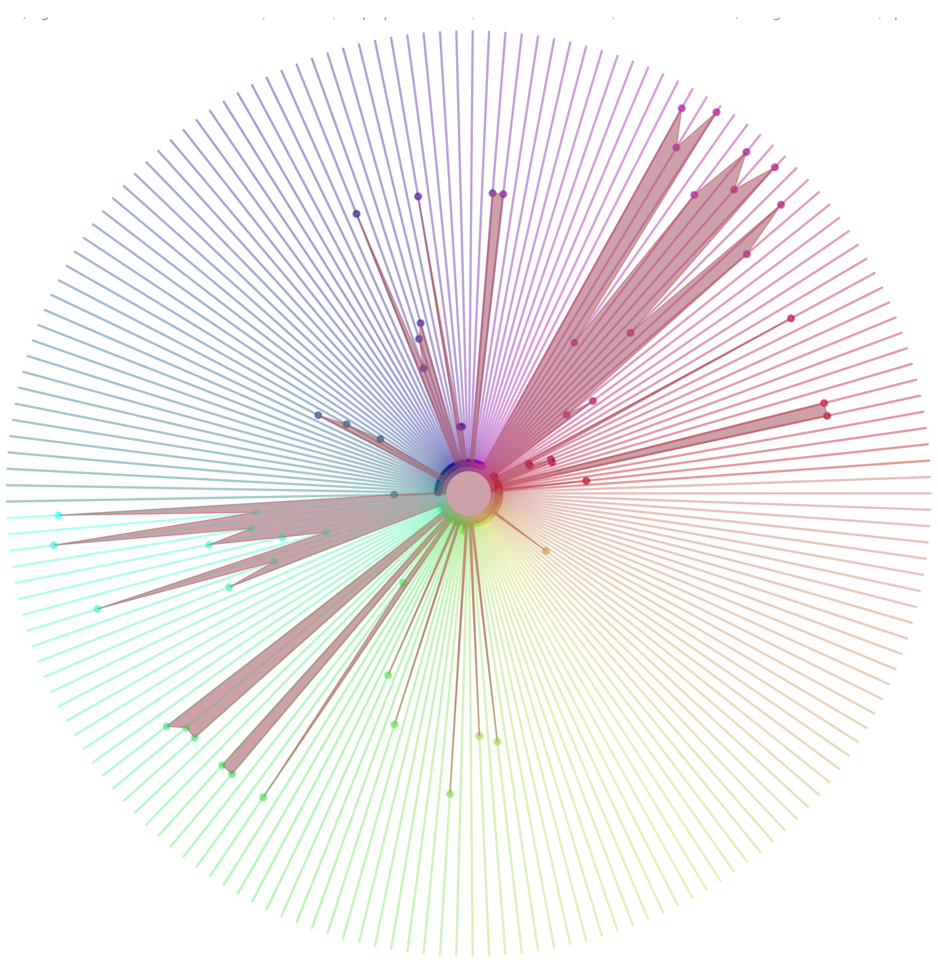


Figure A.21.: Ranking of characteristics for developer *A* in log4j across three clusters (cluster 2, large, original in Figure 4.19b)

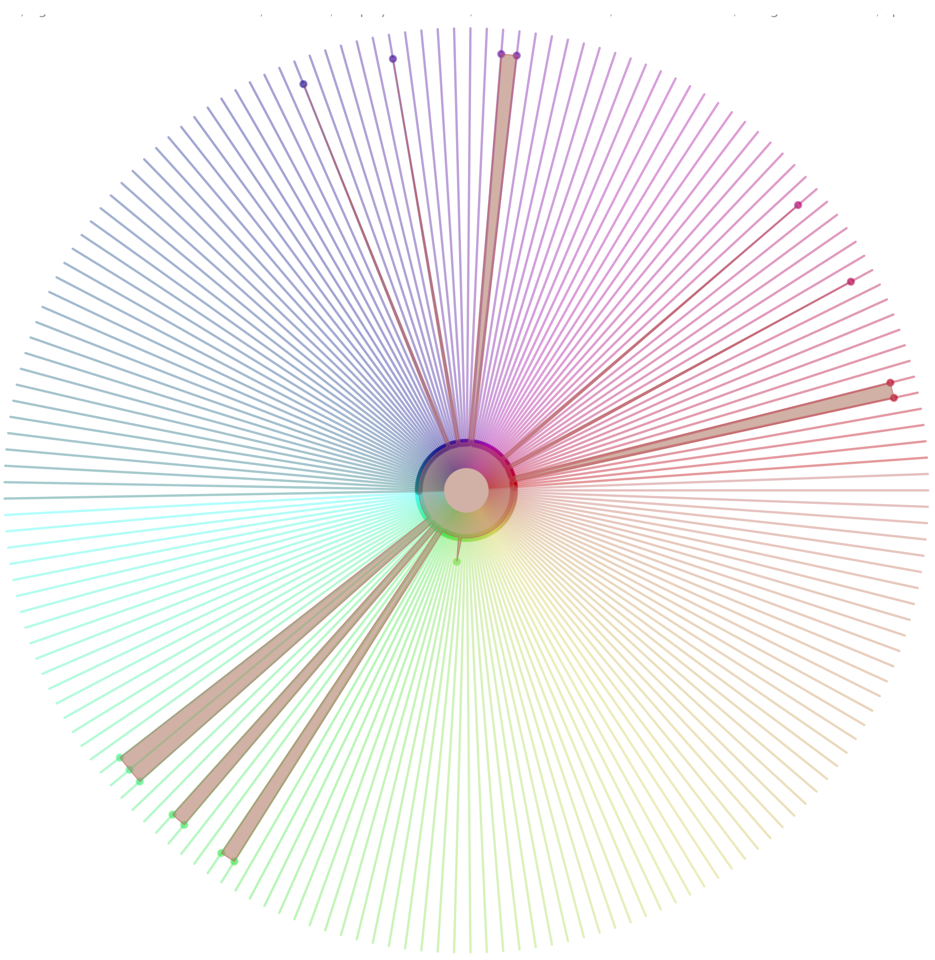


Figure A.22.: Ranking of characteristics for developer *A* in log4j across three clusters (cluster 3, large, original in Figure 4.19c)

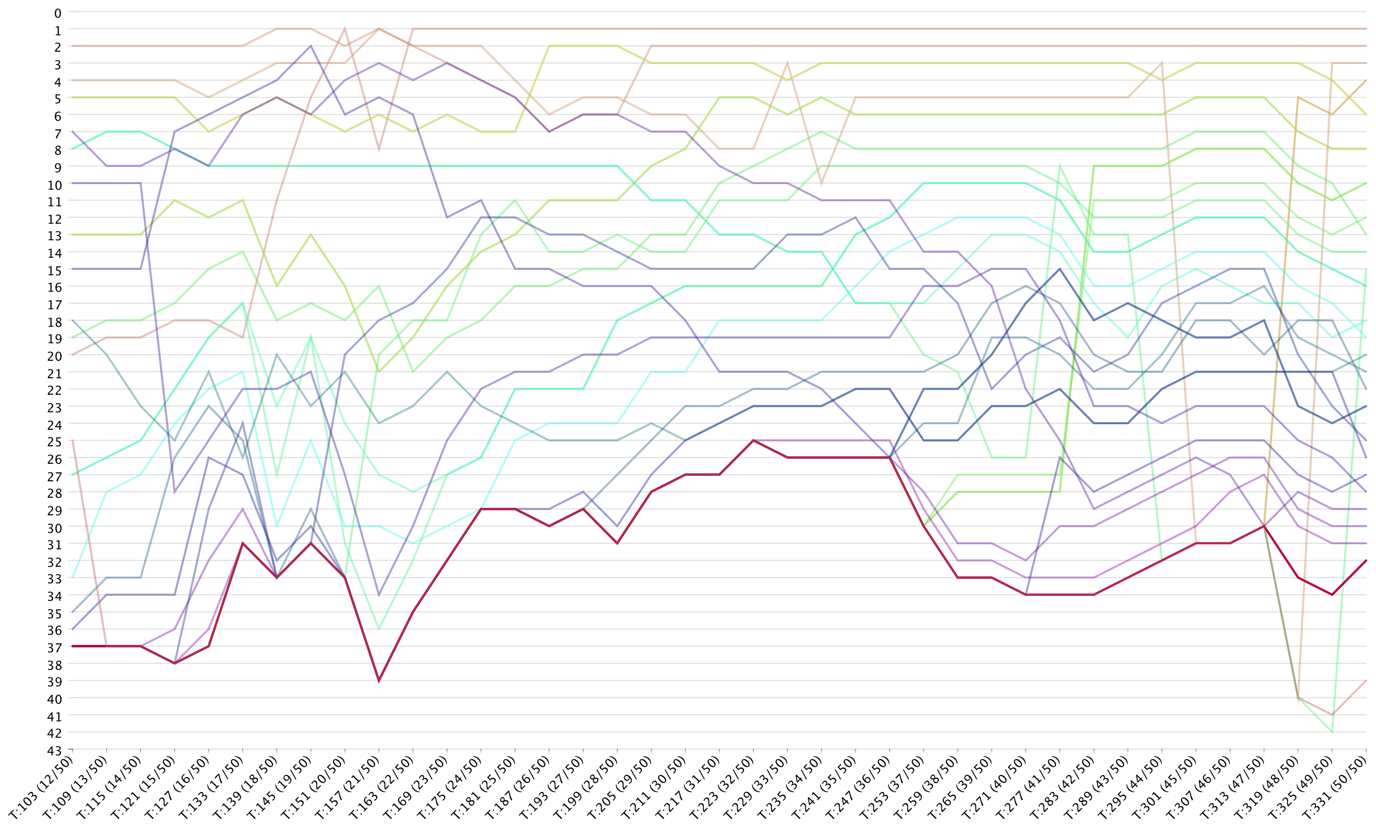


Figure A.23.: Characteristics ranking over time for developer *A* in log4j (cluster 1, large, original in Figure 4.20a)

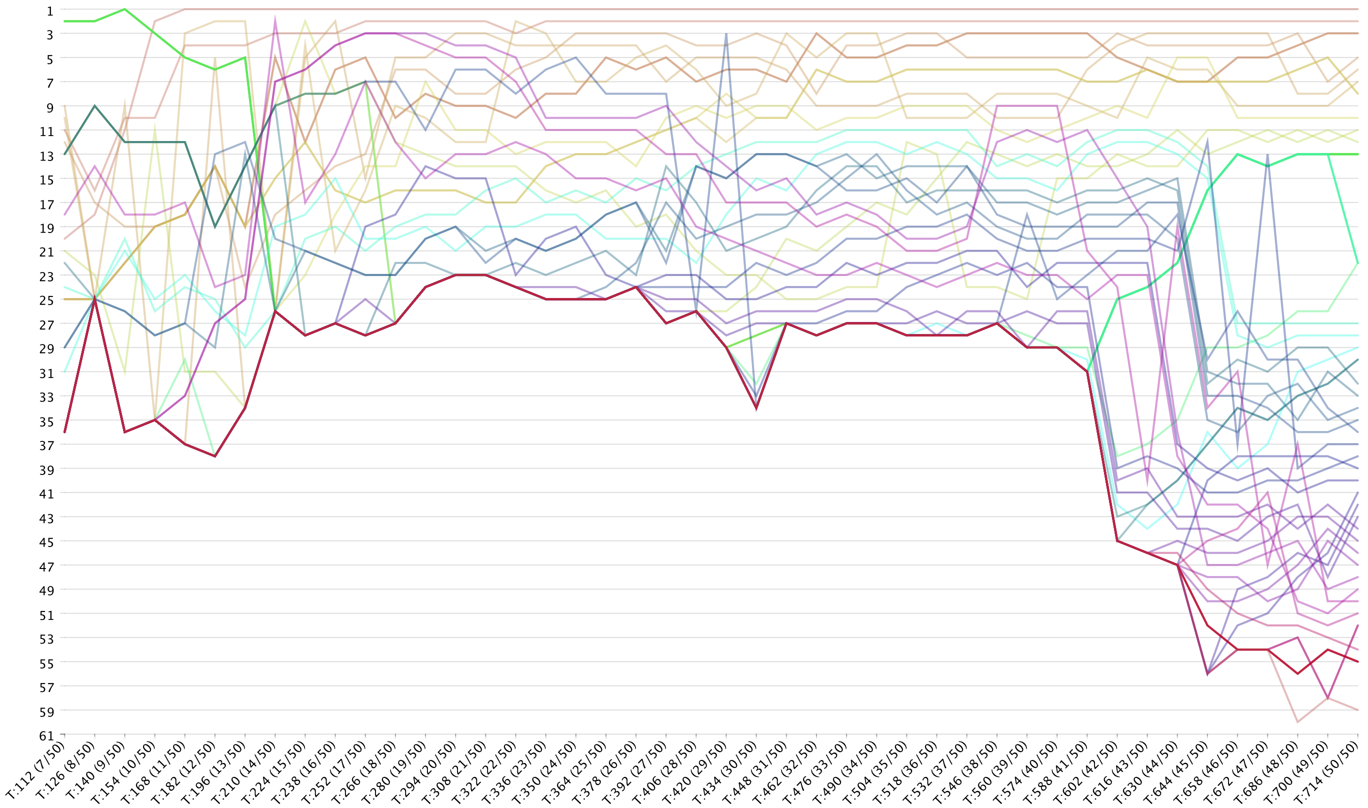


Figure A.24.: Characteristics ranking over time for developer *A* in log4j (cluster 2, large, original in Figure 4.20b)