Using Provenance to Trace Software Development Processes

Master Thesis

Heinrich Wendel

Matr. Number: 2131889

Bonn, 11th of June 2010

in cooperation with
German Aerospace Center
Simulation and Software Technology

University of Bonn
Institute of Computer Science III
Professor Dr. Armin B. Cremers

Statement of Affirmation

I hereby assure that this thesis was exclusively made by myself and that I have used no other sources and aids other than those cited.

Bonn, 11th of June 2010

Contents

List of Figures

List of Tables

List of Listings

1 Introduction: The What and Why

This chapter begins with a description of the relevance of the working environment in which this thesis was written. Afterwards, the detailed topic is explained in the motivation section of this thesis and distinguished from existing approaches. Finally, the approach to handle the topic is outlined and the structure of the work explained.

1.1 Working Environment

This thesis was written at the German Aerospace Center in the department for Distributed Systems and Component Software of the Institute of Simulation and Software Technology.

The German Aerospace Center The German Aerospace Center (DLR) is Germany's national research center for aeronautics, space, energy and mobility. Additionally it is responsible for planning and implementation of the German space programme.

The DLR employs approximately 6500 people in 29 institutes and 13 locations all over Germany: Berlin, Bonn, Braunschweig, Bremen, Cologne (headquarters), Goettingen, Hamburg, Lampoldshausen, Neustrelitz, Oberpfaffenhofen, Stuttgart, Trauen and Weilheim. DLR also has offices in Brussels, Paris and Washington, D.C.

The DLR budget for in-house research and development work and other internal operations amounts to approximately 570 € million, of which approximately half comes from revenues earned by DLR. DLR also administers the space budget of the German government, which totals some 920 € million (2008) [dlrb].

Institute Simulation and Software Technology The mission of the DLR institute "Simulation and Software Technology" (SC) is research and development in software engineering technologies, and the incorporation of these technologies into DLR software projects. Current activities focus on component-based software for distributed systems, software technologies for embedded systems and software quality assurance.

Software development costs account for an increasing fraction of the overall labor cost in research and development projects. For DLR, this fraction amounts to about a quarter of its personnel. As compared to manual techniques which still prevail in software development, the use of state-of-the-art software engineering technologies leads to better project results and more efficient development processes. In cooperation with DLR's engineering institutes SC takes part in demanding software development projects where it complements the application know-how of the other project partners.

Through its cooperation with world-wide leading partners as well as its collaboration in international forums and standardization bodies, SC takes an active role in the development of new software technologies and builds up corresponding expertise at DLR for future projects.

The DLR "Simulation and Software Technology" is located at the DLR sites in Cologne, Braunschweig and Berlin. It is divided into the departments "Distributed Systems and Component Software" and "Software for Space Systems and Interactive Visualization" [dlrc].

Department Distributed Systems and Component Software The Distributed Systems and Component Software department focuses on Grid Computing and on component-based software development for distributed systems. The tasks include the development of modern software

tools engineering applications and software integration systems based on service oriented architecture as well as consulting and development in the field of automation and practical use of software engineering processes. [dlra]

Relevance Although one of the core competencies of the DLR lies in the field of engineering, software is developed by more than 1000 employees [Sch07]. The Department Distributed Systems and Component Software provides the software engineering representative, responsible for setting up DLR wide software engineering standards and supplying all institutes with modern methods, tools and consultancy in the area of software development. On the one hand, research in up-to-date topics in the area of software engineering helps the DLR to maintain a leading role in its main research areas. On the other hand, the amount of software developed at DLR offers a useful test environment for evaluating new concepts in real life scenarios.

1.2 Motivation

Today's software development processes are complex in their nature. Simple development process models, such as the waterfall model [Roy87], have been replaced by iterative methods, such as the Rational Unified Process (RUP) [Kru03], or more flexible agile [Coc01] approaches. Furthermore many tools are used in these processes, such as version control systems, issue trackers or continuous integration frameworks.

In all phases of the software development, from planning to the implementation and final delivery, numerous interactions occur between developers, the tools they use while developing and automatically between different tools. Examples of these interactions include: i) discussion about a feature request, ii) entering or changing requirements in an issue tracking system iii) automatic code style checks during a check-in.

Information about these processes is, if available, distributed over the different tools being used. Version control systems feature a history of all files and their editors, issue tracking systems a list of all comments for an issue. Still, the missing link between these different tools makes it either impossible to draw conclusions from these data, or very time-consuming, considering the immense amount of available data. Many questions appearing on a daily basis cannot be answered, e.g., "To which requirement does this commit belong?" or "Who is responsible for a failing unit-test?".

This master thesis aims at making it possible to follow and reproduce the entire process of developing a software product, being able to easily answer the above questions at any point in the development process. An example process is selected and transformed into a specific model, holding all required data. As this data should not be inserted manually into the model by developers, but recorded automatically, the model must be integrated into a distributed software engineering tool suite (version control system, bug tracking, integrated development environment, etc.). Furthermore, it must be possible to perform automatic audits and analysis on the collected data.

The foundations of this thesis have been established in the research area of provenance. Provenance is defined as the origin or history of data [BKcT01]. General models to describe and store data, query the data for information and methods for adapting processes to the model have been developed and are used as a basis [Mor09].

1.3 Distinction

Aside from the previously mentioned and later explained subject of provenance, the topic of this thesis touches other active areas of research, without directly belonging to them. These include: rational management, traceability and mining software respositories. Furthermore, the Integrated eLearning Environment by Industrial Logic is presented in order to demonstrate a real world application that currently traces some aspects of a software developers work.

Rational Management Rational management aims at recording design decisions during a software development process. These decisions result from the given requirements and the records later help to justify them or reevaluate them with changing requirements or technology. Dutoit et al., e.g., solve this issue by recording a decision using four steps: the question, options, criteria and arguments. Using this structure, and adequate tools, design meetings can be captured and questions of why a given decision was made can be answered at any time of the process. [DPM00]

By connecting two parts of a software development process, the requirements and the design decisions, rational management makes an important contribution to tracing the decision making aspect of the software development process. Still, two major points differ from the proposed approach. First, the question why a decision has been made, is not the focus of the thesis. Although there also have been attempts to answer the "why"-question [BKcT01] in provenance research, this research primary deals with the process itself and the data generated by the process and its history [Mor09]. Second, the thesis aims at an more integrated approach, tracing a complete development process, including, e.g., tests and continuous integration.

Traceability Traceability in the context of software engineering usually refers to requirements traceability. Requirements Traceability deals with tracing the links between requirements, design artifacts, tests and code in both directions [GF94]. Although many methods helping to achieve that have been developed, e.g., the Requirements Matrix [Dav90], the automation level is still very low [KS09]. On the one hand, this results in a large amount of overhead for the developers in addition to the actual development; on the other hand, it is error prone and mistakes will be made. Another recent field handles the recovery of missing traceability information between software artifacts, using information retrieval techniques. [DLFOT06]

Aside from being an active area of research, both of the above methods do not completely cover the proposed topic. Although requirements traceability has been handled in a sufficient way, it primarily focuses on the requirements and does not include the wider scope of the development process. The recovery of traceability information is not a focus of this thesis as it is acted on the assumption that all required data is already collected during the development process and will not be lost.

Mining Software Repositories Recently, the field of mining software repositories has gained great attention [Has08], trying to handle the massive amount of data stored in tools, such as the Concurrent Versions System (CVS) [cvs], Subversion (SVN) [svn] or Git [git]. Although they provide revision control and version history of the managed data, the built-in query mechanisms are quite limited. Recent topics include visualization of changes [LML09], identification of typical usage scenarios of software repositories [HGH08] and analysis of change patterns, tracing defects back to their source and even predicting them [KZPW06].

Although these topics may have similarities to the proposed thesis, some core differences can

be identified. First, the approaches only operate on one tool, a software repository, not on a distributed set of tools. Second a large amount of research is carried out on deriving information that is based on the given data and, therefore, does not always lead to 100% correct results, whereas in a provenance model all data is available and recorded from the beginning, requiring only the correct queries.

Application Lifecycle Management Application Lifecycle Management (ALM) Systems aim to provide an integrated tool that manages artifacts and their relations in every stage of their life cycle, ranging from the initial requirements engineering process, to the final release management. The problem of current ALM systems is that they try to provide all functionality integrated in one big tool. It is not possible to customize the tool for the needs of individual development process [Sch06].

Although ALM systems cover the same scope as this thesis, the complete development process, there are two core differences to the proposal. First, ALM systems integrate all functionality into one unflexible tool, in contrast this thesis connects the existing tools, making it possible to adopt it to the individual software development process. Second, ALM systems do not provide query mechanisms to easily answer questions that may arise during the development process.

The Integrated eLearning Environment by Industrial Logic Industrial Logic is a company offering training, workshops and eLearning in order to teach software development and agile methodologies. The company does no only focus on traditional approaches, teaching developers best practices in presentations and offering written tutorials or web casts, but have also developed more advanced tools to rate the result of the development, as well as the individual steps the developer performed to get these results [Ker09]. Industrial Logic does this by providing plug-ins for commonly used integrated development environments (IDE), such as the Usage Data Collector (UDC) for Eclipse [udc]. These plug-ins monitor changes of source code, the time they occurred, tracking the results of unit test executions and the time needed by a person to fulfill an assignment. This leads to detailed statistics and suggestions how to improve the personal development style.

This case shows an interesting approach of how to track the interaction of a developer and the tools he uses and how to benefit from the results. Furthermore, recording of these data is an automatic process, integrated into the tools. Still, this approach only focuses on a small area, the interaction of the developer with the IDE and does not take into account the rest of the process.

Conclusions Although all of the presented methods handle some aspect of the development process, they always lack important features. None provides a completely integrated view of the entire process, rather they focus only on the requirements (Rational Management) or the integrated development environment (Integrated eLearning Environment). Traceability does not provide the required grade of automation, charging the developer with additional tasks. Finally, mining of software repositories approaches the topic from another view point, not trying to record the complete process, but to reconstruct it based on partial information.

1.4 Approach

This thesis proposes a solution to answer questions such as 'Which requirement causes the most build failures', based on provenance technologies. The first step towards achieving this it to collect questions arising on a daily basis during the development of the simulation framework RCE [SWS09] [rce], developed at DLR, identified during a survey of the development team. Afterwards, the development process is analyzed using the PrIMe methodology [MMG+06], adapted to work with processes instead of applications and the current provenance modeling standard, the Open Provenance Model [MCF+09]. The resulting model is converted into a graph model using the graph database Neo4j [neoe] and the provenance questions are converted into queries using the graph programming language Gremlin [gre]. Finally, the integration of the model into the process and the distributed tool environment, in order to automatically collect the relevant data, is carried out using a service oriented REST [Fie00] architecture.

1.5 Structure

Following this introduction, the master thesis is structured into six additional chapters. It roughly follows a software development process, using the PrIMe methodology to analyze processes in respect to its provenance.

Chapter 2 The second chapter introduces the core concepts. It explains how a software development process works in general, and in more detail at DLR. Afterwards, the state of the art of provenance research is discussed. Finally, the principles of graph databases and graph queries are explained.

Chapter 3 Requirements form the input of each project, resulting in a final product that fulfills all requirements. Requirements in the area of provenance are specified in terms of questions about the process. Chapter three formulates questions on the software development process that arise on a daily basis and the reasoning behind them.

Chapter 4 When the requirements are known, a design is created and an appropriate provenance model is developed. This model is able to hold all necessary data in order to answer the defined questions.

Chapter 5 Based on the model developed in the previous chapter and the evaluation of provenance technologies in chapter two, the concrete implementation, using a graph database, is described. This implementation provides automatically executable queries to answer the stated questions.

Chapter 6 A model is not really useful without real world data. Therefore, the method of how the model can be integrated into a distributed development tool suite is described. Furthermore, the model is filled with collected test data and a performance evaluation is performed.

Chapter 7 Finally, the entire thesis is reviewed again, achieved goals are summarized and open problems outlined.

2 Background: Things to Know in Advance

This chapter explains the background knowledge used throughout the thesis. First the topic that is analyzed, namely software development processes, is introduced in its basic form and more complex realization used in real world projects. Afterwards, the research area of provenance is discussed, which is focused on tracing the history of data in processes. Finally, the basics of graph databases are explained, which are used to store and query the collected process data.

2.1 Software Development Process

The need for structured software development processes date back to the software crisis of the late 1960's. The NATO conference on Software Engineering built the foundations of this field of study [NR68]. The ever increasing complexity of software systems [Dvo01] resulted in a large number of development process models and confirmed the need for a structured development process. An overview is provided by Sommerville [Som07].

2.1.1 Definition

Beginning at the straightforward waterfall model, software development models range from more flexible iterative processes, such as RUP, to agile methods. The wide variety of approaches shows that there is not one process that fits the needs of all projects. Every project needs a customized process, Barry and Lang came to the same conclusion by comparing the processes of more than 1000 software developments [BL03].

Waterfall Model The waterfall model [Roy87], shown in figure 1 is a simple, linear model for software development processes. It is divided into five steps, each step taking the previous step as input: i) analysis of the requirements, ii) design of the software, iii) implementation of the software, iv) testing of the functionality, v) maintenance phase. The name 'waterfall model' origins from the fact that these phases are usually graphically arranged in a cascade. This also symbolizes that there is no possibility to go back one step in the process, the lack of flexibility being the main point of criticism. Some variants incorporated the possibility to incrementally return to the previous step, leading to the basics of an iterative approach.

Strictly linear models are not applicable to more complex projects because of their lack of flexibility. This is because it is impossible to finish one phase of a software development project completely before entering the next phase. E.g., it may not be possible to catch all requirements in all their details or the requirements may change.

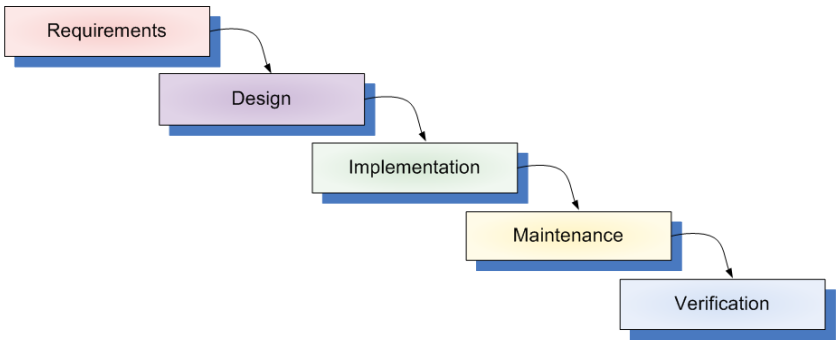

Figure 1: The waterfall model.

Rational Unified Process Due to the shortcomings of the waterfall model, more flexible, iterative approaches, such as RUP [Kru03], outlined in figure 2, have been proposed. Aside from the previously mentioned core steps of the waterfall model, supplemented by business modelling, four phases are modelled orthogonally: i) inception, ii) elaboration, iii) construction, iv) transition. Each of the phases contains each step varied in length. Furthermore, the phases are divided into multiple iterations. This leads to multiple passes of each step, resulting in intermediate results, so called milestones. One advantage of this process is the possibility to adapt to changing or unclear requirements.

One of the main points of criticism of RUP is its complexity and formalism, which is strengthened by the three additional disciplines: i) project management (e.g., time plans, risk management), ii) configuration and change management (e.g., document structuring), iii) environment (e.g., development tools, server, etc.).

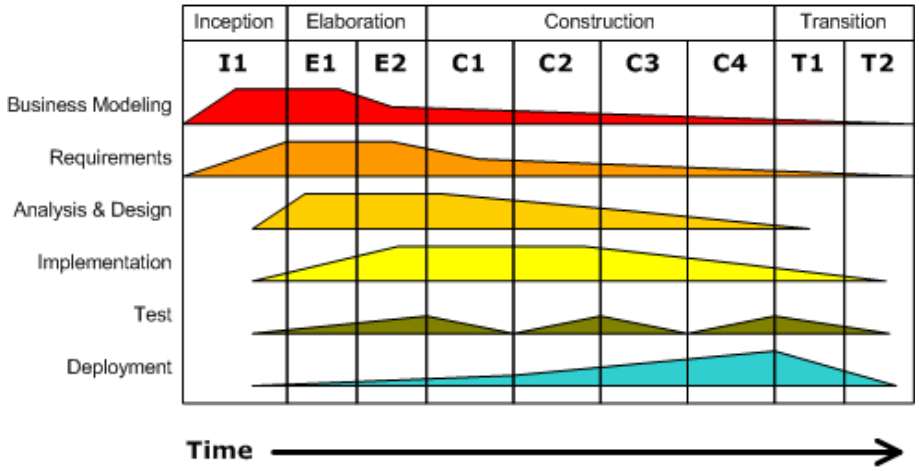

Figure 2: The Rational Unified Process, phases and disciplines. [Kru03]

Agile Methods In recent years new models arose that tried to solve these issues, they are commonly summarized as agile processes [Coc01]. The agile manifesto [BBvB+01] formulates four main principles: i) Individuals and interactions over processes and tools, ii) Working software over comprehensive documentation, iii) Customer collaboration over contract negotiation, iv) Responding to change over following a plan. Thus they focus on reducing the formal processes as much as possible, focus on actual work, keeping close contact to the customer and being as flexible as possible on changes.

Conclusions It can be said that there is not one software development process that is applicable to every project. Each project needs to adapt one of the processes as a base, depending on size and complexity, for example, and customize it to match its special requirements.

2.1.2 Typical Software Development Process at the DLR

The typical software development process of the DLR, outlined in figure 3, was designed by the Institute for Simulation and Software Technology to be individually customized for each project.

Being an iterative model, the process is divided into three cycles. The outer cycle defines the release process of the whole system, the middle cycle concentrates on individual components

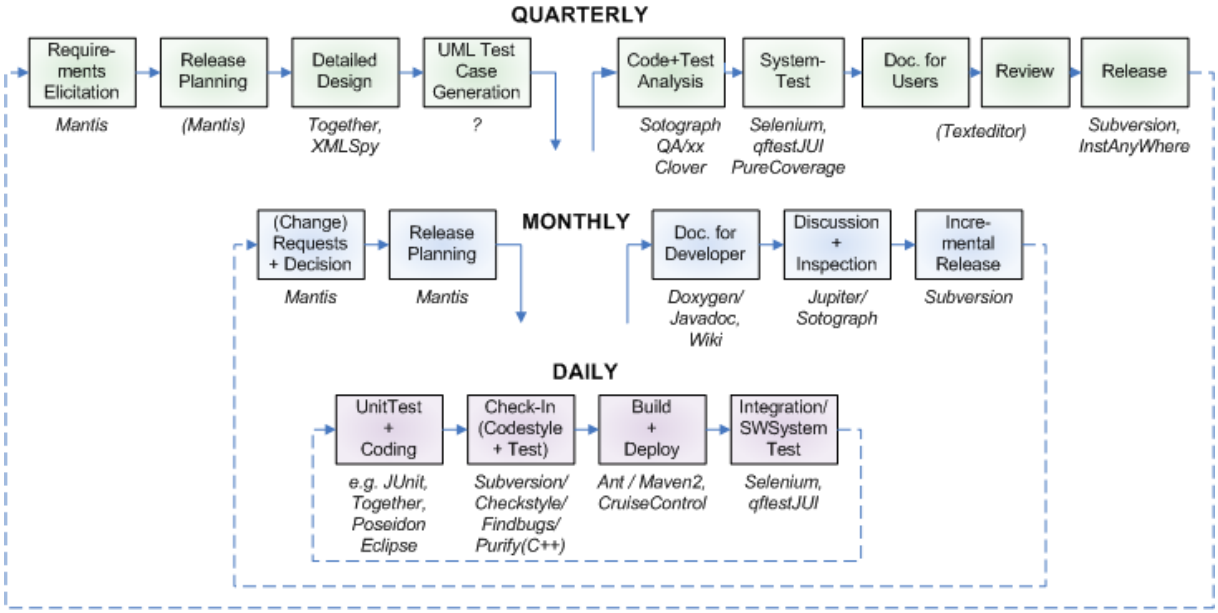

Figure 3: Typical DLR software development process. [BHS09]

and the inner cycle describes the daily work of a software developer.

In the first phase, it is important to review the given requirements (Requirements Analysis) and choose the ones to implement for the release (Release Planning). Afterwards, a concept (Detailed Design) can be created for each requirement and test cases have to be created (Test Case Generation), in order to later validate the implementation. Before the release is created, the code is reviewed on a higher level against selected rules (Code + Test Analysis) and the test cases are run (Code + Test Analysis). Documentation on how to use the software has to be created (Doc. for Users) and after a final review of the whole product (Discussion - Review), the actual release is created (Release).

The second phase begins with a more detailed planning of the selected requirements ((Change Request) + Decision / Release Planning) on a subsystem basis. After the implementation in phase three, documentation for developers have to be created (Doc. for Developer) and the code is reviewed by other developers (Discussion + Inspection). Afterwards, the subsystem is released (Incremental Release).

The implementation phase begins with writing unit tests and the actual code (UnitTest + Coding). After a check-in to the central repository, which performs automatic code style- and verification tests, (Check-In - Codestyle + Test), the continuous integration system attempts to build the project and deploys it as snapshot version (Build + Deploy). Finally, tests that test the functionality of the entire component, have to be created (Integration / SWSystem Test).

Based on the needs of each software project, steps in the software development process can be added or removed, tools selected and the length of the iterations adapted.

2.1.3 Adaption to the Development of the RCE

This chapter explains how the individual steps of the software development process are executed in the project Remote Computing Environment [SWS09] [rce], an integration platform for distributed simulations in the engineering domain.

- *Requirements Elicitation / (Change) Requests + Decision:* Based on experiences from previous iterations and user requests, new requirements and bugs arise. All issues are collected in a central *issue tracking* system, namely *Mantis* [man]. Validity, state and priority of all issues are discussed in weekly meetings, called Jour fixe.
- *Release Planning:* Based on the previous discussion, the project leader creates a release plan, managed by the *issue tracker*. Issues are assigned to individual developers.
- *Detailed Design:* If necessary, design meetings are called up during which part of the team discusses how to implement a given issue. Both, meeting minutes and design documents, are uploaded to the project's homepage [rce], the central place for all *documentation*. The homepage is realized using the *MoinMoin Wiki* [moi] engine.
- *Unit Tests + Coding:* The main development language of RCE is Java. The *Wiki* contains documentation on how to set up the complete development environment based on *Eclipse* [ecl], including *JUnit* [jun], *Maven* [mav], *Checkstyle* and *Cobertura* [cob].
- *Check-in:* All code is stored in a central *SVN* repository. The mainline of the code is called trunk. More complex issues are developed in branches, so they do not conflict with the stable code in the trunk, and merged back to the trunk when completed. Issues are related to check-ins via their identifier, which has to be provided in the commit message.
- *Codestyle Tests:* During check-in to the trunk, *Repoguard* [LPR+09], integrated as *SVN* hook script, checks the code against the style conventions using *Checkstyle* [che] and rejects it on failure. Furthermore it is possible to restrict files access to certain developers or groups.
- *Build + Deploy: Continuous Integration* is handled by *Hudson* [hud]. After each commit and on a nightly basis the trunk of the entire project is *built* and results reported by e-mail. Moreover all *unit tests* are run and *code coverage* reports are generated. After a successful build, the created snapshot are uploaded to the local Maven repository managed by Nexus [nex].
- *Developer Doc:* Developer *documentation*, intended for developers integrating applications into the framework, is generated in two ways. First, the code comments are automatically converted into an API reference by Javadoc. Second, guides are written, containing information and examples of how a component can be used.
- *Discussion + Inspection:* When a developer feels a that a feature is finished, a code review is performed. Other developers examine the code and try to identify problems. Code reviews are logged in the *Wiki*, and from the results new issues are created in the *issue tracker*.
- *User Doc.:* User *documentation* is written for two different types of users. Administrators, setting up new instances of RCE, are supported by an administration guide that explains how to install, configure and manage RCE instances. End users are provided with documentation on how to use the graphical components of RCE.
- *Release:* Before the actual release, a new tag is created in SVN and a branch created for minor maintenance releases. A *release script* creates the final archive files, which, together with the created documentation and samples, are uploaded to the project's site on sourceforge [rce]. Finally, an announcement is sent to the mailinglists informing subscribers about the changes in the new release.

Tool Support The description of the development of RCE shows that a multiple tools are used throughout the process. All of the tools involved and the connections between them are displayed in figure 4. The central part of the development remains the developer itself, supported by his IDE. He interacts with the issue tracker to enter new issues, changes details of existing issues and closes fixed ones. The main tool used for storing documentation is the Wiki. While coding, the developer regularly builds the code, runs unit tests and checks their code coverage. This is also carried out by the continuous integration system after check-in. During check-in, the repository checks to which issue the check-in belongs (by examining the comment) and validates the coding style and permissions on files. Finally, the user triggers the release script, which automatically checks out the code, builds it and creates the actual release.

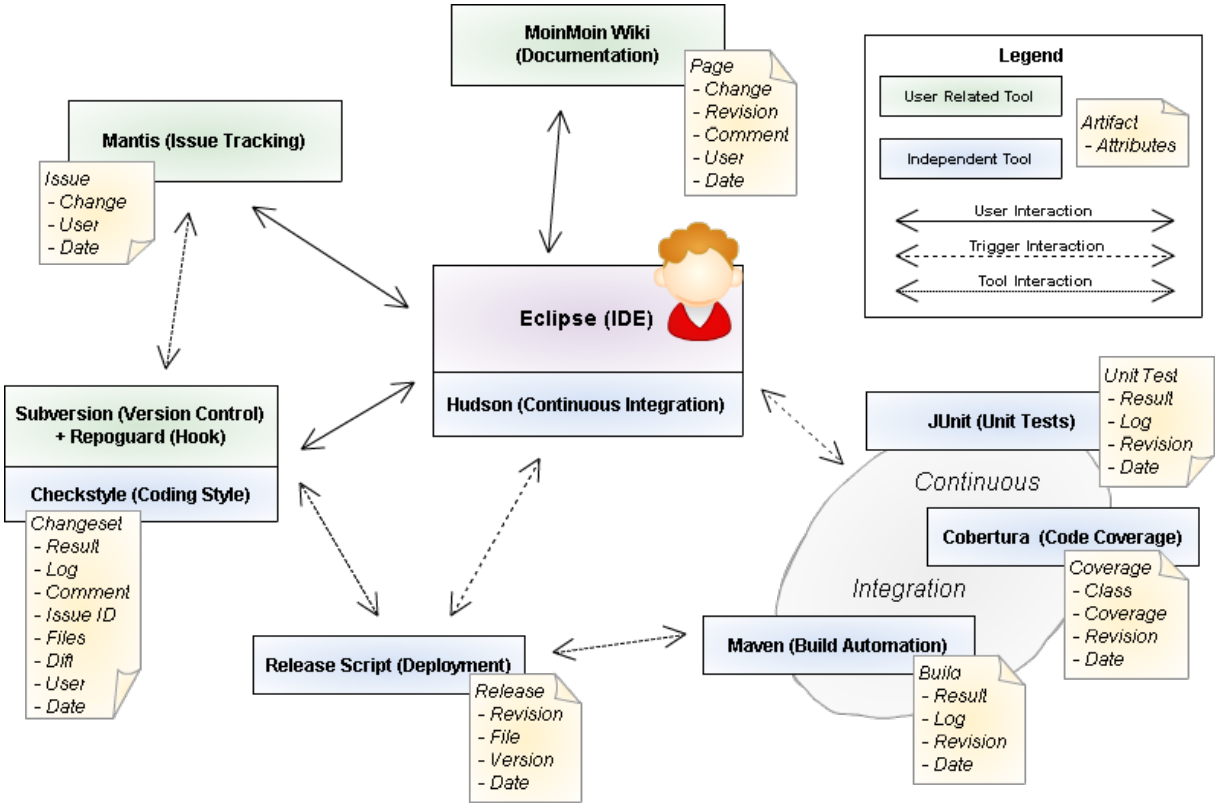

Figure 4: Tools of the development environment used for developing RCE.

2.2 Provenance

This section shows the current state of research in the field of provenance. After providing a definition of provenance and describing the concepts around it, the Open Provenancen Model (OPM) is introduced. The OPM provides a standardized model that tries to connect all existing approaches. Afterwards, PrIMe is presented, a method to analyse applications in order to make them provenancen-aware. Finally, the current approaches for recording provenance information in real world applications are evaluated and an appropriate technique selected for the implementation.

2.2.1 Definition

There exists several analogous definitions for provenance. The Merriam-Webster Online Dictionary defines provenance as:

> *(i) the origin, source; (ii) the history of ownership of a valued object or work of art or literature.*

Generally, provenance refers to the documented history of an object. This documented history helps to trace how the object was created, e.g., how simulation results were determined or how documents have been assembled [MGM+08].

After taking into account different definitions of provenance in literature, Moreau summarizes them the following way [Mor09]:

> *The provenance of a piece of data is the process that led to that piece of data.*

Applying this definition to the provenance of a software development process, it refers to the documentation of the entire process, taking into account the generated artifacts (requirements, source code, tests, etc.) and the interaction between the different actors (developers, tools, etc.). This documentation then allows questions such as "which developer commited the code resulting in this unit test failing?" to be answered.

2.2.2 Concept

The research behind provenance includes multiple topics. Simmhan et al. derived a taxonomy summarizing the most important topics, pictured in figure 5. This thesis touches many of the areas shown in this image. It examines (Subject of Provenance) the software development process (Process Oriented) in order to (Use of Provenance) answer certain questions about it (Data Quality / Audit Trail / Informational). The scheme used for storing the data is the Open Provenance Model (Provenance Representation / Annotation). The implementation (Storing Provenance) is based on Neo4j and information extracted (Provenance Dissemination) using the Eclipse plug-in Neoclipse (Visual Graph) and the query language Gremlin (Queries). Furthermore, a REST API is provided (Service API) to insert new data into the model.

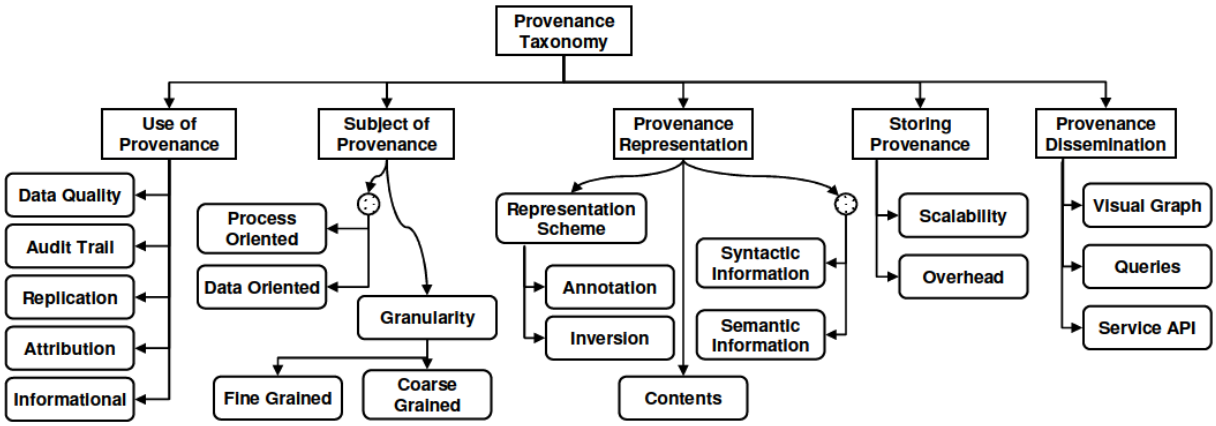

Figure 5: Provenance taxonomy. [SPG05]

Groth et al. describe the general architecture of a provenance system [GJM+06]. This can be summarized into a simpler view, shown in figure 6. The central part of this architecture is a provenance store, which is provided with a provenance model by an administrator. Respecting

this model, applications can now begin to record process data and insert it into the store. This allows users to query, reason and visualize the collected data.

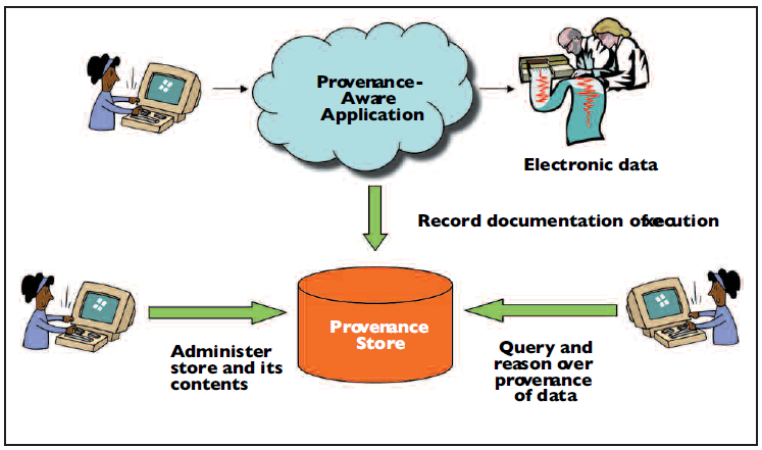

Figure 6: Provenance life cycle. [BKcT01]

2.2.3 The Open Provenance Model

The OPM [MCF+09] was developed in order to standardize the different modelling approaches into one common format. This format can be used to interchange data between different implementations.

The base of the OPM is a causality graph, which is a directed and acyclic. It is directed because each edge in the model contains a direction and acyclic because of processes happening one after another on the timeline. The graph can be annotated in order to add additional information.

Nodes There are three different types of nodes: i) artifacts, ii) processes, iii) agents. In the graphical representation, artifacts are displayed by ellipses, processes by rectangles and agents by octagons. The nodes are defined as follows in the OPM specification:

- *Artifact*: Immutable piece of state, which may have a physical embodiment in a physical object, or a digital representation in a computer system.
- *Process*: Action or series of actions performed on or caused by artifacts, and resulting in new artifacts.
- *Agent*: Contextual entity acting as a catalyst of a process, enabling, facilitating, controlling, or acting its execution.

Edges Edges represent the causal dependencies between the nodes they connect, the source nodes depict the effect, destination nodes the cause. There are five different provenance dependencies, all shown in figure 7: i) used, ii) wasGeneratedBy, iii) wasControlledBy, iv) wasTriggeredBy, v) wasDerivedFrom. The edges are defined as follows in the OPM specification:

- *Used*: A "used" edge from process to an artifact is a causal relationship intended to indicate that the process required the availability of the artifact to be able to complete its execution. When several artifacts are connected to a same process by multiple "used" edges, all of them were required for the process to complete.
- *WasGeneratedBy*: A "was generated by" edge from an artifact to a process is a causal relationship intended to mean that the process was required to initiate its execution for

the artifact to have been generated. When several artifacts are connected to a same process by multiple "was generated by" edges, the process had to have begun, for all of them to be generated.

- *WasControlledBy*: An edge "was controlled by" from a process P to an agent Ag is a causal dependency that indicates that the start and end of process P was controlled by agent Ag.
- *WasTriggeredBy*: An edge "was triggered by" from a process P2 to a process P1 is a causal dependency that indicates that the start of process P1 was required for P2 to be able to complete.
- *WasDerivedFrom*: An edge "was derived from" from artifact A2 to artifact A1 is a causal relationship that indicates that artifact A1 needs to have been generated for A2 to be generated. The piece of state associated with A2 is dependent on the presence of A1 or on the piece of state associated with A1.

The relationship can be defined in more detail using *roles*, which are textual descriptions, e.g., a WasDerivedFrom relationship between a release and a revision could be named as 'build from'.

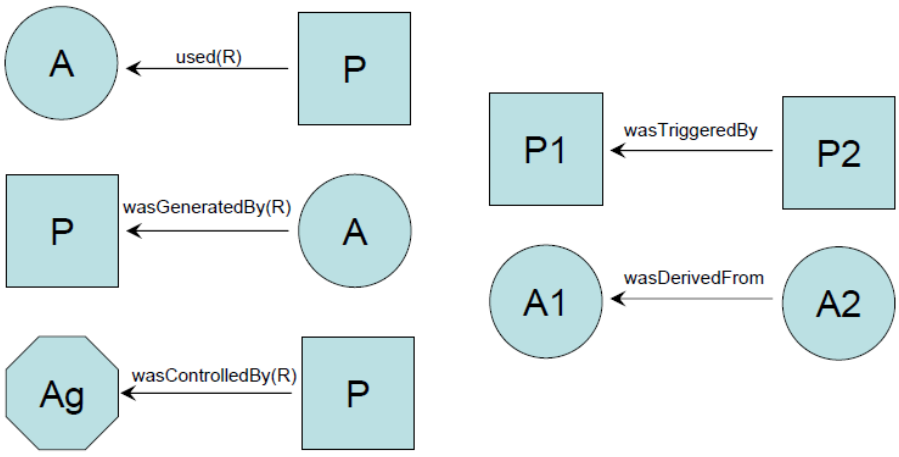

Figure 7: Nodes and edges in the Open Provenance Model. [MCF+09]

Annotations In order to add more information to the graph it can be annotated. Annotatable entities are: i) the whole graph, ii) individual nodes, iii) edges, iv) annotations itself. Annotations are defined as a list of property-value pairs.

Example Figure 8 shows an example provenance graph, recorded how John baked a Victoria Sponge Cake. WasDerivedFrom edges are represented by solid lines, all others by dotted lines. The *bake* process was controlled by the agent *John*. The ingredients used to bake such a cake are *100g butter*, *100g sugar*, *two eggs* and *100g flour*, resulting in the final *cake*, generated by the bake process. Special roles are defined on the edges connecting the bake process and the ingredients and between the final cake and the ingredients. Finally, two examples for annotations are provided, the cake is of *quality yummy* and the flour of *type raising*.

Further Topics The OPM also specifies more advanced topics. *Profiles* allow to group annotations for reuse in different models. *Inferences* allow to find sources of data generation even though they were not explicitly modelled. *Temporal constraints* can be identified by looking

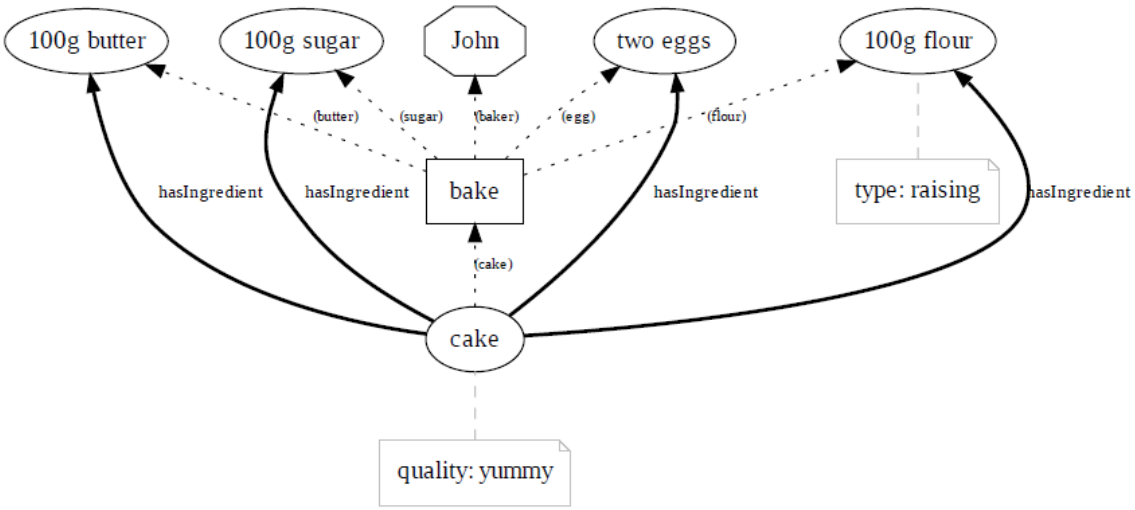

Figure 8: OPM Example - The Victoria Sponge Cake Provenance. [MCF+09]

at the relationships. Finally, *accounts* can be defined to specify different recording granularity levels. These topics are not relevant for this thesis, thus it is only referred to the specification for further details.

2.2.4 PrIMe

PrIMe [MMG+06] was developed to provide engineers with a methodology to analyze processes and make them provenance-aware, i.e. record provenance information in order to later answer questions about the process or analyze it for errors.

Overview The overall structure of PrIMe, shown in figure 9, is divided into three phases, which can be roughly characterized as: i) defining the requirements, ii) conception of a model, iii) implementation. The conception phase is carried out iteratively until the required detail of information has been reached. Each phase is further divided into multiple steps:

Phase 1 During the first phase, the questions that shall later be answerable using the collected data are compiled (step 1.1). Afterwards, they are analyzed by the data items and scope is necessary to answer them (step 1.2). The result of this phase is a textual description of the questions, accomplished by: i) a start item, i.e. the item of which the query will be sought, ii) the scope, identifying the important parts of the application that lead to the start item, iii) the processing steps required to acquire the final answer.

Phase 2 In the second phase, a list of actors that produce the identified data items is created (step 2.1) and, based on the application structure, the interactions between the individual actors, i.e. the data flow, is analyzed (step 2.2). The interactions can be visualized using so called interaction graphs. Afterwards, it can be identified which actors are important for the derivation of a data item, so called knowledgeable actors (step 2.3) and which need additional adaption as it is not possible to collect their data, so called hidden actors (step 2.4).

Phase 3 Finally, phase 3 deals with exposing data items, which are currently not available, via wrappers around hidden actors (step 3.1) and adding the actual provenance functionality to the application itself (step 3.2).

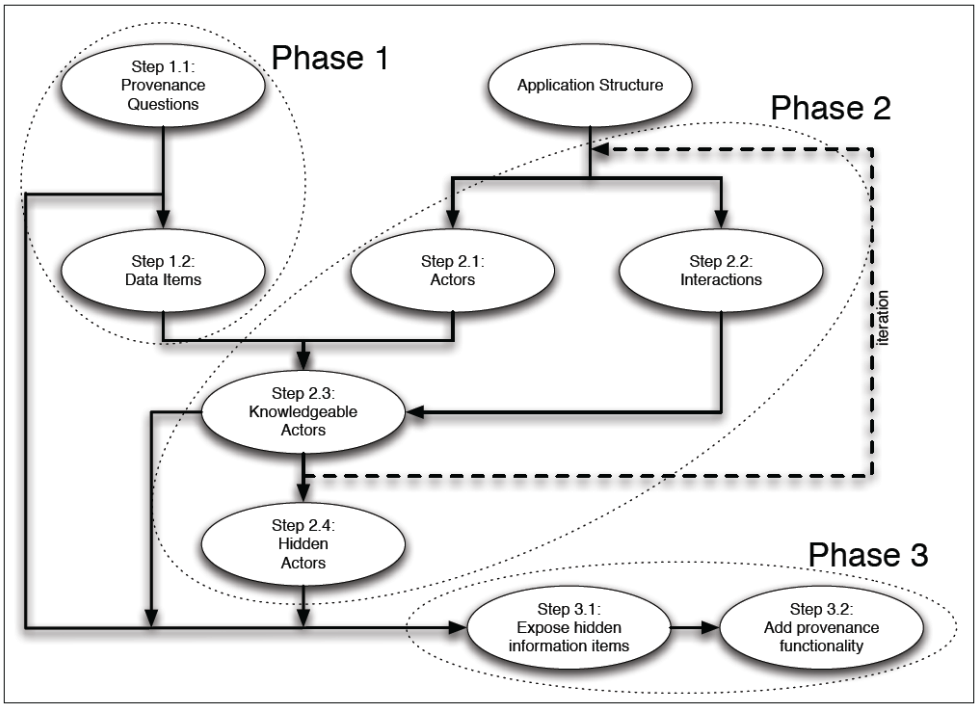

Figure 9: Overall structure of PrIMe. [MMG+06]

Adaption Some adaptions have to be made to PrIMe to make it applicable for this thesis:

- Originally PrIMe was developed using the p-assertion protocol [GMT+06] as a base model for recording provenance information. Instead of this, the Open Provenance Model, representing the state of the art, is used in this thesis.
- PrIMe was developed as a methodology to make applications provenance-aware. In the case of a process, this entire process has to be seen as being the application.

Based on the assumptions that all information is available in a software development process via the tools and is without hidden actors, the following structure is used:

- *Questions:* This step is equal to phase 1 of PrIMe; all provenance questions that should be answerable on the process are described using the same tabular representation.
- *Actors:* Afterwards, the process is analyzed for individual actors, in the case of an application, these are components, in the case of a process, these are sub-processes. This represents PrIMe step 2.1.
- *Data Items:* For each of these sub-processes, the relevant input data items (represented by the used relation in OPM) and the output data items (represented by the wasGeneratedBy relation in OPM) are identified and displayed using OPM graphs.
- *Interactions:* Afterwards, the interactions between the individual actors are analyzed, according to step 2.2 of PrIMe, again using the OPM as a graphical representation.
- *Final Model:* The knowledge gained in the last three steps is compiled into the final model.
- *Implementation:* Finally, the implementation of the model and adaption of the process is outlined, aligning to PrIMe phase 3.

2.2.5 Realisations

The recording of provenance information in real world processes does not only require methods, such as PRiME, to analyze the process and data models, such as OPM, to store the data. The models also have to be implemented, based on the available technologies in computer science. The provenance community regularly arranged provenance challenges in order to obtain an overview of the different approaches and compare these approaches in a given scenario [MLA+08]. Furthermore, roundup papers have been published for the same purpose, e.g. Simmhan et al. provide an overview over the approaches of the e-Science community [SPG05] and Holland et al. do a comparison based on the querying technology [HBM+08]. Holland et al. distinguished four general approaches to store and query the data:

- *Relational:* The most common type of databases are relational databases, they store the data in tables with rows and columns and build connections through relations between tables. Data can be inserted and queried using the widely accepted SQL standard. [EN03]
- *XML and XPath:* XML provides a format for data exchange that can be read by both machines and humans. It describes a metalanguage, which can be used to structured text in a hierarchical form. The data can then be queried using the XPath language. [HWR+07]
- *RDF and SPARQL:* The semantic web, an extension of the world wide web that attempts to create meaning of data usable to computers, defined by the Resource Description Format (RDF), is often serialized using XML. As the name "web" already implies, information in RDF is structured in a net or graph and is not hierarchical. This graph structure can then be queried using the SPARQL language. [dVGT09]
- *Semistructured:* Semistructured data is defined by a system of objects, holding attributes and connections between them, without any underlying formal structure. This concept is not directly tied to any technology, but can be realized in different ways, e.g. using objects of programming languages and the according query languages, such as LINQ in .NET. [Tro07]

The results of the third provenance challenge [cha], which specifically focuses on OPM, show a strong preference using semantic web technologies. This has the advantage that many existing tools, middlewares or graphical interfaces can be reused. On top of these existing tools, such as Sesame [ses] or Jena [jen], interfaces especially designed for provenance information have been designed, e.g., Tupelo [MFG+09] and the OPM Toolbox [opm].

Still Holland et al. argue that the underlying hierarchical XML structure of RDF and the inflexibility of the SPARQL query language do not make it the best choice for storing provenance information. Therefore, they propose to use semistructured data, which has a graph like structure as its base. Based on previous research in this area, they created a new query language PQL. Unfortunately, the language is currently only usable in combination with their provenance system, which records file system actions.

As previously mentioned, the semistructured approach is generally technology independent. Another database, implemented in Java, storing graphs, that has successfully been used to store provenance information [TC09] is Neo4j [neoe]. In addition there exists the graph programming language Gremlin [gre], which is based on XPath. Gremlin can be used to query graphs stored in Neo4j.

2.3 Graph Databases

In recent years, the research in the area of databases moved away from all purpose relational or object oriented databases, towards more specialised databases adapted to the use case. Four different database types emerged (i) key-value stores, (ii) column stores, (iii) document databases, (iv) graph databases. Graph databases are especially suited to store data that already has a graph like structure, such as relations occurring in social networks. These databases natively store graphs with their nodes and edges and offer querying technologies [Eif09].

2.3.1 Neo4j

Neo4j [neoe] is one of the most advanced graph databases available. It is written in Java, but provides bindings to other programming languages as well. One main focus of the developers of Neo4j was to keep the database small, simple and fast. It can be used as embedded database with a footprint of only 500k. One instance can handle several billion nodes, relationships and properties, an even greater number through scalable replication and partitioning concepts. Stability has been proven through 6 years of production use. More advanced features cover transaction management, domain models, graph traversal APIs and connections to semantic web technologies, such as SPARQL. Neo4j is dual licenced under AGPLv3 and commercially.

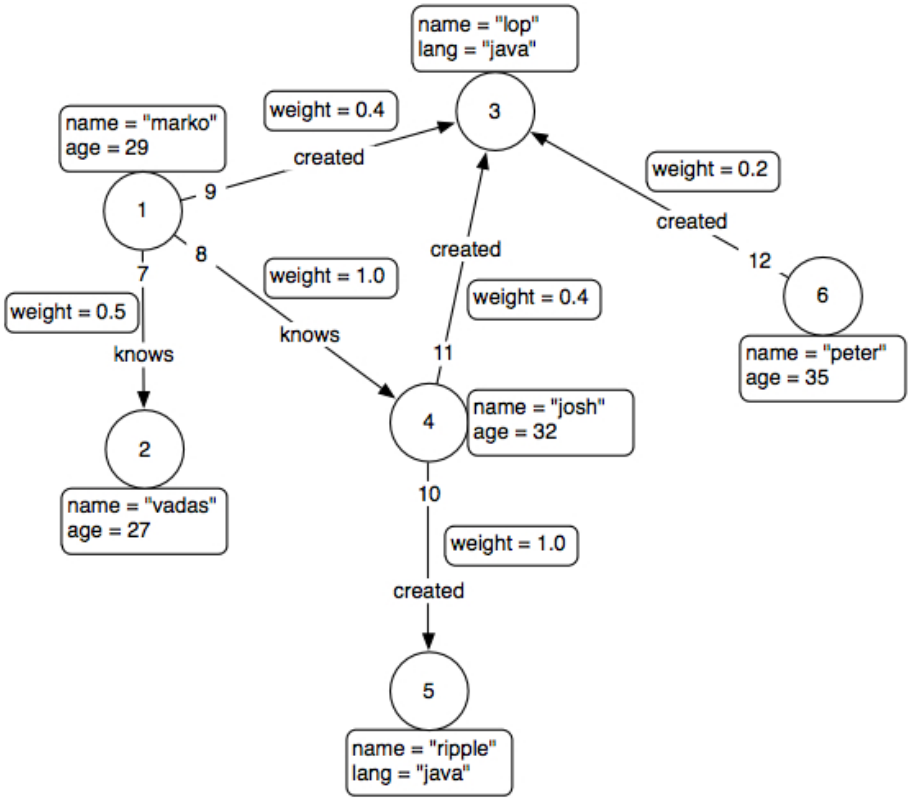

Figure 10: Example of a simple property graph. [gre]

Property Graph Model The structure of a graph in Neo4j instance does not follow any defined schema, it is semistructured, following only the rules of a simple property graph that is displayed in figure 10. A property graph is a graph that is: i) key/value-based, ii) directed, iii) multi-relational. Key/value-based refers to the fact that all vertices and edges can hold any number of properties, defined by key/value pairs. Directed signifies that each edge has a direction from

one edge to another associated to it. Multi-relational denotes that there can exist more then one edge between two vertices.

A property graph has a set of vertices where each vertex has: i) a unique identifier, ii) a set of outgoing edges, iii) a set of incoming edges, iv) a collection of properties defined by a map from key to value.

Furthermore, a property graph has a set of of edges; each edge has: i) a unique identifier, ii) an outgoing tail vertex, iii) an incoming head vertex, iv) a label denoting the type of relationship between its two vertices, v) a collection of properties defined by a key/value map.

Indexing In order to search for vertices in a performant manner, indexing structures are used. Neo4j does not provide an implementation for an indexing mechanism, rather an API that can be used by such services; the most common being Apache Lucene [luca].

Indexing is not performed automatically when assigning properties, rather the developer has to manually specify each key/value to be indexed. It is important to note that, although lucene also supports full text indexing, only one-dimensional indexes are supported. This means if a search is carried out to locate, for example, a vertex with two special properties, one index should be generated using the values of both properties.

Neoclipse Neo4j features an Eclipse plug-in, named Neoclipse [neog]. This plug-in helps to visualize, navigate and filter graphs, as well as edit vertices or edges; figure 11 shows an example screenshot. The upper part of the image shows the graph, centered at the *Users* vertice. The depth of visible vertices, from the center vertice, can be configured; in this example it is three. By double clicking a vertice it becomes the new center and the displayed nodes around the center are reloaded. Vertices can be assigned different icons based on their properties. The properties of a vertice or edge are shown in the bottom left of the image, after is is selected. The bottom right window allows the user to filter out certain relationships from the view.

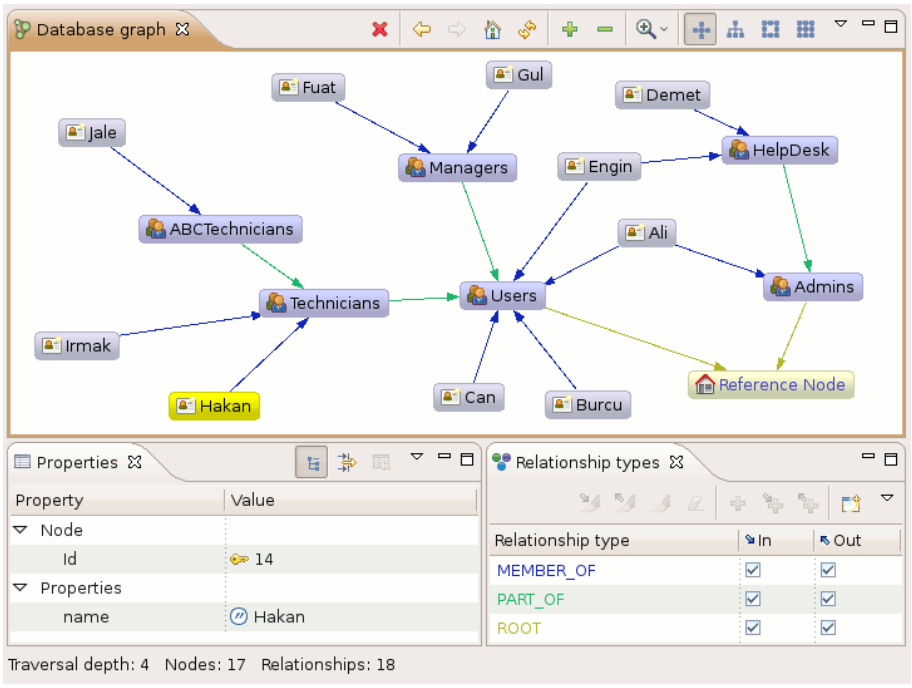

Figure 11: The Neo4j Eclipse plug-in Neoclipse. [neoe]

2.3.2 Gremlin

Gremlin [gre] is a graph-based programming language. It allows users to query graphs, as well as explore and edit them. Gremlin is not bound to any underlying graph database. A graph can be loaded from multiple backends, one of them being Neo4j. Basic graph traversal is handled using XPath 1.0 expressions, all of its core and mathematically functions can be used. As XPath 1.0 is not a touring-complete language, some features have been added, e.g., language statements, such as assignments, loops or conditions. Additionally it is possible to extend the library with own functions.

Vertices as modelled as element nodes, properties as attribute nodes. Additionally, the following nodes have been added to support graph navigation: i) *outE*, to select all outgoing edges of a node, ii) *inE*, to select all incoming edges of a node, iii) *bothE*, to select all edges of a node, iv) *outV*, to select all outgoing vertices of an edge, iv) *inV*, to select all incoming vertices of an edge, v) *bothV*, to select all vertices of an edge.

Gremlin can either be used via the provided console or integrated into Java. After issuing a command, the console displays the intermediate results, which greatly aids in working with graphs. Listing 1 shows some basic example queries. For a complete reference of all features it is referred to the Gremlin Wiki pages [gre].

```
1  // Load a graph
2  gremlin> $_g := tg:open()
3  ==>tinkergraph[vertices:0]
4  gremlin> g:load('data/graph-example-1.xml')
5  ==>true
6  gremlin> $_ := g:id('1')
7  ==>v[1]
8
9  // Select all outgoing edges
10 gremlin> ./outE
11 ==>e[7][1--knows-->2]
12 ==>e[8][1--knows-->4]
13 ==>e[9][1--created-->3]
14
15 // Select all outgoing vertices of those edges
16 gremlin> ./outE/inV
17 ==>v[2]
18 ==>v[3]
19 ==>v[4]
20
21 // Show names of all known people with age > 30.
22 gremlin> ./outE[@label='knows']/inV[@age > 30]/@name
23 ==>v[3]
24 ==>v[5]
```

Listing 1: Basic Gremlin examples.

3 Requirements: Asking Questions

Many problems can occur throughout this complex process. The questions presented in the following have been identified during an internal survey of the RCE team. Each of the questions following in this chapter has been assigned to one of the following purposes stated in brackets. Although, in practice, a question can be assigned to more then one purpose, it is assigned to a main category to provide an example.

- *Error Detection:* During day to day development, builds or unit tests may suddenly fail. In such cases it is important to identify the source of the error. In many cases this may be the responsible developer, who has the knowledge to fix the problem. It may also be the failure of a tool, e.g., occurring after an upgrade. (17, 18, 19, 20)
- *Quality Assurance:* Customers are always interested in a product with maximal quality. Quality assurance, therefore, has always been a very important topic, not only in the domain of software engineering. The number of unit tests or code coverage percentages give important hints on where the quality of the product is very high or still deficient. (2, 3, 6, 15)
- *Process Validation:* Following a defined process is another way of performing quality assurance. By following norms such as ISO 9001 the quality of the final product is not assured, rather the assurance lies in the quality of the process that led to the product. This is especially important in the area of medical software, where a process validation is required. (4)
- *Monitoring:* Often problems do not become visible until closer inspection of the project. Automatic monitoring and notifications can help to identify these problems, e.g., to see if an issue takes longer to implement than expected. (14)
- *Statistical Analysis:* Statistics help to interpret and draw conclusions out of collected data. They can be used by managers, e.g., to decide if the project is in time, needs more resources or if developers can be put into another project. Developers are interested in statistics to see how they perform or compare to others. (1, 11, 12)
- *Process Optimization:* Another use of monitoring, error detection and statistics is the optimization of the process itself. E.g., many commits related to one issue may show that the issues should have an increased granularity in the next iteration. A build tool that fails often because of segmentation faults, may be replaced by a new one. (8, 13)
- *Developer Rating:* There is no widely accepted method to rate the productivity of developers and it may not be a popular topic. Still the collected data can provide some hints in order to decide which developer to assign to a specific problem. The data may indicate that some developers are better in writing unit tests and some in documentation. This helps to improve the process. (5, 7, 9, 10)
- *Informational:* Sometimes data has to be collected for informational purposes, e.g., when creating a release announcement it shall contain a list of all bugs fixed. (15, 16)

Not all possible questions can be addressed in this thesis, therefore, a categorization and example questions for all categories have been determined. These questions shall help to prove the general feasibility of provenance for software development processes and identify the important data to store in the model, e.g., the name of the developer is likely of interest, in contrast to the size of

his shoes. For real world usage the model can be extended with additional data, following the modelling approach used for this thesis.

Two main categories of questions have been identified: i) single tool questions, depending only on data of a single tool of the development process and ii) multi tool questions, depending on the connection of the data of multiple tools.

3.1 Single Tool

Single tool questions can be answered without recorded provenance information, by querying the database of the individual tool holding the required data. Although this can be accomplished, the questions must also be answerable with the help of the proposed provenance technology. Keeping all this data in a single repository also provides the advantage of being able to use one single query technology, independent from the queried repository. The single tool questions are divided into i) simple questions and ii) aggregated questions.

3.1.1 Simple

This type of questions is characterized by having a very limited scope and requiring only a very small subset of the data to be answered. They are usually answerable by single line queries.

1.) Who is responsible for implementing issue X? This question only deals with the data of the issue tracking system. Each issue has one developer assigned to it and sometimes this person has to be identified, e.g., because somebody has to implement an issue that depends on the issue in question and wants to plan his time.

Start item	The latest version of issue X.
Scope	Only the issue.
Processing step	Return the assignee field.

Table 1: PrIMe analysis of provenance question 1.

2.) What is the current build status of the project? This question only deals with the data of the automatic builds. To obtain a quick overview of the project status, it is often helpful to see if it currently builds. A build may take a long time, which can be reduced by doing automatic builds and only querying the output.

Start item	The latest build.
Scope	Only the build.
Processing step	Return the exit code of the build.

Table 2: PrIMe analysis of provenance question 2.

3.) What is the current overall code coverage? This question only deals with the data obtained in code coverage reports. Code coverage is an indicator of the code quality. Continuous integration systems can be configured to automatically generate such reports during the builds.

Start item	The latest code coverage report.
Scope	Only the code coverage report.
Processing step	Return the percentage of overall coverage saved in the report.

Table 3: PrIMe analysis of provenance question 3.

4.) From which revision was release X built? This question only deals with the data of the automatic builds. There exists no software that is 100% free of bugs. Therefore, it is often important to identify from which revision in the version control system a release was built in order to provide a bug fix in all later revisions.

Start item	The release with the version number X.
Scope	Only the release.
Processing step	Return the revision number associated with the release.

Table 4: PrIMe analysis of provenance question 4.

3.1.2 Aggregated

In contrast to simple questions, the following questions usually require special aggregation capabilities in the query language, known from relational databases. Examples for aggregations include counting and finding the average, minimum or maximum of a given set.

5.) How many unsuccessful commits did user X do? The version control system performs the same code checks as the local development environment and rejects commits on errors. A high number of failing commits may indicate an incorrectly configured development environment.

Start item	The user X.
Scope	From the user, to his commits.
Processing step	Return the count of the unsuccessful commits.

Table 5: PrIMe analysis of provenance question 5.

6.) How did the number of unit tests change in the last month? Quality assurance is not a single step in the development process that occurs at a specific point in time, but rather a continual process. This questions helps to provide an indication of how the quality of the software changed over time.

Start item	The latest unit tests report.
Scope	From the latest unit tests report, to the one a month ago.
Processing step	Return the difference between the two numbers.

Table 6: PrIMe analysis of provenance question 6.

7.) Which developer is most active in contributing documentation? Documentation is one of the most essential steps in software development. Therefore, it may provide a motivation to reward positive results in this area.

Start item	All documentation items.
Scope	From the documentation, to the users.
Processing step	Aggregate the documentation items by developer and return the one with the maximum.

Table 7: PrIMe analysis of provenance question 7.

8.) How many releases have been produced this year? The productivity, growth or stagnation of a project may be judged by the number of releases produced in a given time.

Start item	All releases.
Scope	Only the releases.
Processing step	Limit the releases to those produces in the last year.

Table 8: PrIMe analysis of provenance question 8.

3.2 Multi Tool

The real advantage of the approach of this thesis has to be validated by answering multi tool questions. These questions can only be answered by connecting the data of at least two tools, recorded in a common provenance graph. Three subcategories of questions have been identified: i) developer related, ii) requirements related, iii) error related.

3.2.1 Developer Related

The main focus of the following questions are the developers. These questions identify the role of the developer in the process and connections he makes between different tools and data items.

9.) How many issues were implemented by developer X for release Y? The number of implemented issues may be a measurement of the productivity of a developer, but it may also provide an indication of the difficulty of his tasks.

Start item	The release for which the statistics are queried.
Scope	From the release, to the issues closed for it.
Processing step	Limit the issues to those implemented by developer X.

Table 9: PrIMe analysis of provenance question 9.

10.) How many commits did developer X contribute to release Y? The number of commits may be a more appropriate indication of a developer's productivity. However it may only show the inefficiency of his work.

Start item	The release with the version number Y.
Scope	From the release, to the issues fixed in it to the related commits.
Processing step	Limit the commits to those performed by developer X.

Table 10: PrIMe analysis of provenance question 10.

11.) How many developers contributed to issue X? Many developers contributing to a single issue may indicate that the issue is too large and should have been divided into multiple issues.

Start item	All versions of issue X.
Scope	From the issue, to all related commits and documentation.
Processing step	Find and count the developers that changed the issue, performed commits and created documentation.

Table 11: PrIMe analysis of provenance question 11.

12.) Which developers contributed to release X? Contributions to a release can occur on different levels. Developers work on an issue, commit code or write documentation.

Start item	The release with the version number Y.
Scope	From the release, to its issues and related commits and documentation.
Processing step	Output the names of all developers performing these actions.

Table 12: PrIMe analysis of provenance question 12.

3.2.2 Requirements Related

Requirements are a central part of each software development process. They build the foundation for all following steps, implementation, testing and documentation and release. The following questions identify the connections between the requirements and the data items produced by these steps.

13.) How many commits were needed for issue X? One indicator for the complexity or importance of an issue may be the number of commits required for its implementation. If the issue is too complex, it may be better to divide the issue into multiple parts next time.

Start item	All versions of issue X.
Scope	From the issue, to all commits related to it.
Processing step	Return the count of commits.

Table 13: PrIMe analysis of provenance question 13.

14.) How much time has been spent implementing issue X? On the one hand, the time needed to implement an issue may be another indicator of its complexity, on the other hand, a long implementation time may also reveal that the issue was of low priority.

Start item	The latest version of issue X.
Scope	The first version of issue X.
Processing step	Return the difference between open and close date.

Table 14: PrIMe analysis of provenance question 14.

15.) What documentation belongs to issue X? Finding the documentation related to a given feature is often a difficult and time consuming task, which can be simplified by this query.

Start item	All versions of issue X.
Scope	From the issue, to all related documentation .
Processing step	Return all documentation items.

Table 15: PrIMe analysis of provenance question 15.

16.) Which features are part of release X? The announcement of a new release often contains a list of new features. This list can be compiled automatically. Sometimes it is also important to examine the features planned for the release and identify those that are finished and those that are not.

Start item	The release with the version number X.
Scope	From the release, to all issues assigned to it.
Processing step	Return all issues.

Table 16: PrIMe analysis of provenance question 16.

3.2.3 Error Related

Many small problems occurring in day to day development take a large amount of time to fix, as their root cause is not obvious and difficult to detect. The following questions focus on identifying the source of the problems, making them easier to fix.

17.) Who is responsible for reducing the code coverage? The code coverage dropped below the required percentage and now it is important to identify if this is a reasonable exception or if the developer did not implement more unit tests.

Start item	The specified code coverage report.
Scope	From the report, to the associated commit.
Processing step	Identify the developer that performed the commit.

Table 17: PrIMe analysis of provenance question 17.

18.) Which requirement causes the most build failures? The number of build failures caused by a requirement, i.e. caused by committing code implementing a requirement, may be another measurement for the complexity of an issue. It may also show that the requirement was not specified or designed well enough.

Start item	All issues.
Scope	From the issues, to the related commits and their automatic builds.
Processing step	Find all build failures and aggregate them by the issue, finally return the maximum.

Table 18: PrIMe analysis of provenance question 18.

19.) Which changeset resulted in more failing unit tests? Dependencies between issues are often not visible upon first sight, but are revealed if they require changes in other areas of the application. This is especially important if unit tests fail after a commit.

Start item	The specified unit test report.
Scope	From the test report, to the commits.
Processing step	Find all changesets between the specified and the last successful report.

Table 19: PrIMe analysis of provenance question 19.

20.) Which version of maven caused the build failure of revision X? Often it is not the developer causing a problem, but the tools used for development. This kind of questions helps to measure the reliability of a tool.

Start item	The revision with the number X.
Scope	From the revision, to the associated build.
Processing step	Return the version number of maven used for the build.

Table 20: PrIMe analysis of provenance question 20.

4 Concept: The Open Provenance Model

In this chapter a specialized scheme is developed, based on the Open Provenance Model, to answer the stated questions, using the adaption of the PrIMe methodology described in chapter 2.2.4. First, the actors are identified, after which their data items, split into input and outputs are described. Afterwards, interactions between the actors can be identified leading to a compilation of all data in the final model.

All graphical representations can be seen as schemes for domain models, based on the Open Provenance Model. This differentiation has to be made because there is only one Open Provenance Model, instances of this model have to be filled by actual provenance information.

The diagrams use the OPM icons to display agents, processes and artifacts, additionally visualized by the colors green, yellow and red. Annotations are displayed using balloons, containing either an (M) for mandatory or an (O) for optional. Processes always hold the additional time property, indicating the end time of the execution of the process. The relation types are not explicitly stated, but implicitly provided by the OPM types of the start and end vertices.

4.1 Actors

Based on PrIMe Step 2.1 the actors are identified. In the software development process the actors can been seen as the individual steps or sub-processes that are performed by the user or automatically by tools. Although the users theirselves seem to be actors at first sight, they are only data items in the definition of OPM. Derived from the questions, the explanation of the software development process and the tools used, the following sub-processes have been identified:

Issue Tracking The *issue tracking* process consists of all parts that are performed using the *issue tracking tool*. These are: i) request for new features, ii) entering and editing of bugs, iii) release planing by assigning features and bugs to releases.

Development After the implementation of a feature or fixing of a bug, a *commit* to the central *source code repository* is performed. During the check-in, checks for, e.g., coding guidelines are performed and the commit is rejected when errors are detected.

Continuous Integration On a regular basis, the *continuous integration system* automatically i) *builds* the code base and saves the result (success/failure) and the output, ii) runs all *unit tests* and generates a test report, iii) saves the result of the *coverage report*, which is created while running the tests.

Documentation The different kinds of *documentation*, meeting logs, architecture & design documents, code reviews, user, development and administration documents are managed in the *central documentation system*.

Release Finally, a *script* builds the final *release* and prepares all tools for the next release process, e.g., by creating branches and tags in the repository or marking a release as finished in the issue tracker. Afterwards, the script uploads the final archives to a file server.

4.2 Data Items

For each of the identified sub-processes, this step, according to PrIMe 2.2, identifies the data items it uses as input and generates as output. The inputs and outputs are summarized in tabular form. Additionally, one part of the complete scheme is created for each actor. The tables and diagrams help to identify the interactions between the actors for the next step. It is also important that: first, the individual actors may have more inputs and outputs and second, many more agents, i.e. tools controlling the process, may be involved, but only the ones that are relevant to answer the given questions are described.

Issue Tracking Table 21 explains all data items relevant for the issue tracking process. The user creates an issue change which contains all fields that shall be changed in the issue, whether this is an issue or a bug. Based on the previous version of the issue, if it exists, and above changes a new issue is created and assigned to the given release. If the release does not exist it is created. The distinction between issue change and issue is necessary to answer different questions.

Inputs	
User	Every issue change is performed by a user, identified by his name.
Issue Change	The changes to the issue are saved in an issue change item. As not all parts have to be edited at once, everything but the identifier is optional.
- Identifier	The identifier of the issue that was edited.
- Status (O)	The new status of the issue in textual form.
- Assignee (O)	The name of the new assignee of the issue.
- Release (O)	The identifier of the new release to which this issue shall be assigned.
Outputs	
Issue	Each issue change creates a new issue object, derived from its previous version.
- Identifier	The identifier of the issue.
- Status	The status of the issue in textual form.
- Assignee	The name of the assignee of the issue.
Release	The name of the release to which this issue belongs.

Table 21: Data items relevant for the issue tracking process.

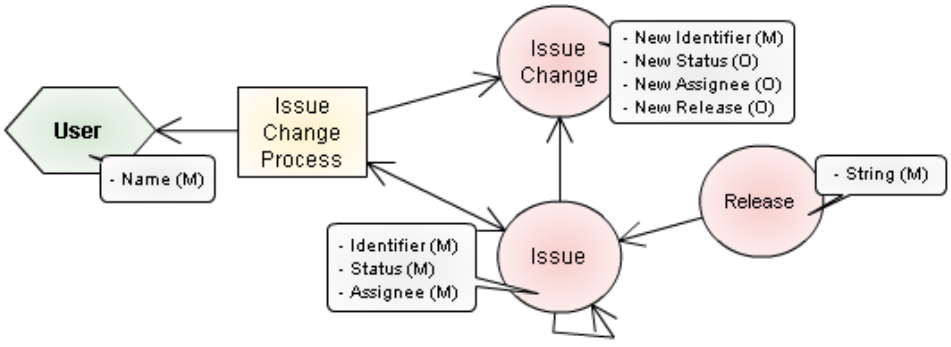

Figure 12: OPM scheme for the issue change process.

Figure 12 shows this process, its inputs and outputs, in an OPM scheme. The user agent controls the issue tracking process. The process uses the issue change artifact and the previous version

of the issue (if existent) as input to generate a new issue, derived from both, and a release (if not existent). It is importance to note that, although the assignee of an issue is a user, user and issue cannot be connected, as OPM does not support connections between agents and artifacts. Therefore, the assignee has been modeled as annotation to the issue artifact. Although a bug contains much more then the three fields status, assignee, release, additional fields are not necessary to answer the given questions.

Development Table 22 explains all data items relevant for the commit process. A commit of new code (the changeset) is triggered by a user who has to enter the corresponding issue identifier in the commit message. Afterwards, the commit is either rejected, if it contains errors, or a new revision is created. Only errors resulting from coding violations are considered, other errors, e.g., merging conflicts, are not handled. The changeset could also hold the complete diff or a list of affected branches, but this is not necessary to answer the given questions.

Inputs	
User	Every commit is performed by a user, identified by his name.
Issue	Every commit belongs to one issue, specified by the identifier of the issue in the commit message.
Changeset	The changeset contains all changes that shall be performed in the repository, including the commit message.
Outputs	
Commit Failure	If the code checks fails, a commit failure occurs and the reason is given in its output.
Revision	If the commit succeeds, a new revision, identified by a revision number, is created.

Table 22: Data items relevant for the commit process.

Figure 13 shows this process, its inputs and outputs, in an OPM scheme. The commit process is controlled by the user and uses the changeset and the given issue identifier as input. Either a commit failure or a new revision is generated as output, derived from the changeset.

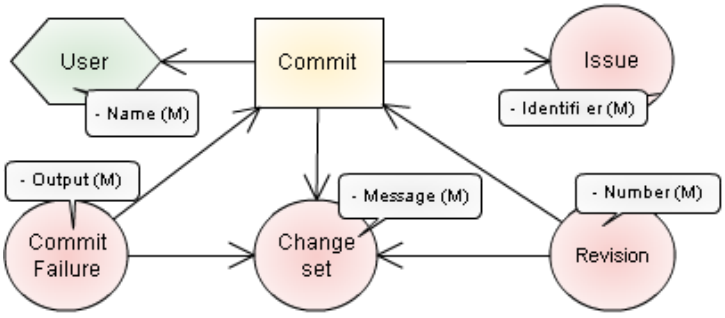

Figure 13: OPM scheme for the commit process.

Continuous Integration Table 23 explains all data items relevant for the continuous integration process. This process can be split into three sub-processes i) build, ii) unit test, iii) coverage, but are combined because of the common modelling approach. All of the sub-processes take a specified revision from the repository as input and generate their results based on this revision: the build output, the test results (number of succeeding and failing tests) and the coverage

report (percentage of covered code). Although these reports can be modelled in much more detail, e.g., by saving the complete build output or detailed coverage numbers for each class, this is not necessary to answer the given questions.

Inputs	
Revision	The continuous integration system takes the code of the current revision as input.
Maven	The maven version used to build the application.
Outputs	
Build Result	The exit code of the build process (0 for success, >0 for failure).
Test Result	The output of the unit tests run, i.e. the number of tests that succeeded and the number of tests that failed.
Coverage Result	The output of the coverage run, i.e. the percentage of code that is covered by the tests.

Table 23: Data items relevant for the continuous integration process.

Figure 14 shows this process, its inputs and outputs, in an OPM scheme. The three subprocesses of continuous integration are modelled as OPM processes. They use the revision artifact, generated by the commit process, as input to generate their results as output, which are directly derived from the used revision. Additionally, the maven tool, which controls the build process, is modelled as agent, containing its version number. It is explicitly required to answer one of the questions.

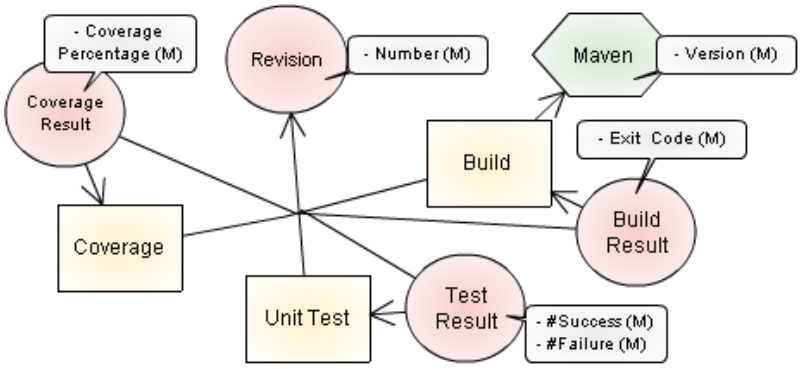

Figure 14: OPM scheme for the continuous integration process.

Documentation Table 24 explains all data items relevant for the documentation process. In contrast to the issue tracking example, the questions cover documentation changes but not the final documentation, this is reflected in the model. The user creates new documentation, specifying the issue to which it belongs, resulting in a documentation change.

Inputs	
User	The name of the user that changes the documentation.
Issue	The issue to which this documentation belongs.
Outputs	
Doc Change	The change to the documentation that has been made, consisting of the page name and the message that was entered as comment.

Table 24: Data items relevant for the documentation process.

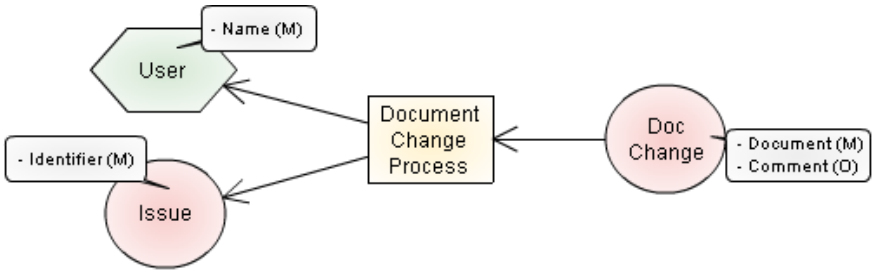

Figure 15: OPM scheme for the documentation process.

Figure 15 illustrates this process, its inputs and outputs, modelled in an OPM scheme. The documentation process, controlled by the user, uses the given issue identifier as input and generates a documentation change. The documentation change contains the name of the documentation item and the comment or reason for the change, provided by the user.

Release Table 25 explains all data items relevant for the release process. The release script creates one or more new archives based on the current revision in the repository and the specified release in the issue tracker.

Inputs	
Revision	The revision from which this release has been build.
Release	The name of the created release.
Outputs	
Archive	The archive that has been generated as result of the release process.

Table 25: Data items relevant for the release process.

Figure 16 shows this process, its inputs and outputs, in an OPM scheme. The release process uses the current revision as input and generates one or more archives which are derived from the given release in the issue tracker.

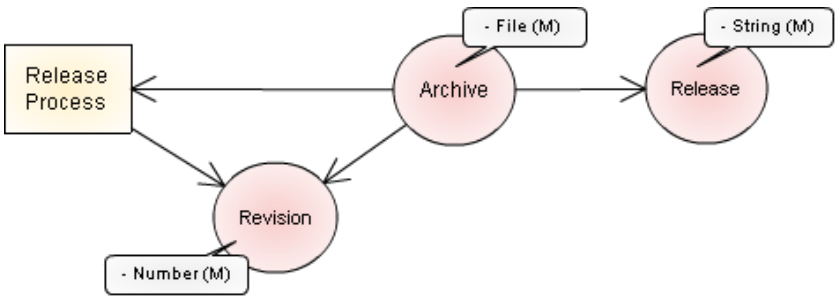

Figure 16: OPM scheme for the release process.

4.3 Interactions

This section deals with the second part of PrIMe step 2.2, the identification of the interactions between the individual actors, based on their data items. The collection of all data items in the last section shows four items which are used by more then one tool: i) user, ii) issue, iii) revision, iv) release.

User The user is involved in three processes: editing issues, committing new code and writing documentation. This is shown in figure 17.

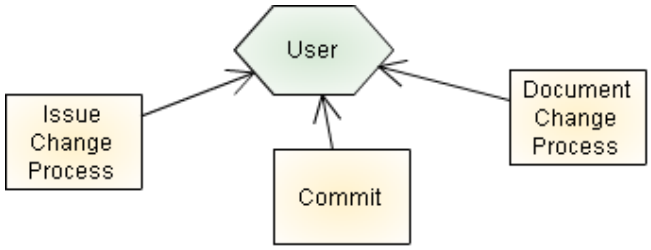

Figure 17: OPM scheme showing all processes controlled by the user.

Issue Issues are generated by the issue change process. Afterwards, they are used by both, the commit and the documentation process, in order to specify the issue to which the created code and respectively documentation belongs. The results of these two processes are not derived from the issue, because the issue is only used for documentation purpose. This is shown in figure 18.

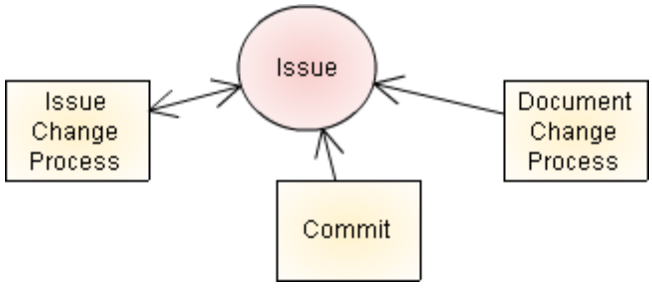

Figure 18: OPM scheme showing all processes interacting with the issue artifact.

Revision Revisions are generated by the commit process. Afterwards, they are used by the continuous integration tools to check if the build still succeeds, the unit tests execute and the code coverage is satisfying. Finally, the release process uses a revision to create a new release archive. All of the results are directly derived from the revision as they are executed on the code contained in the revision. This is shown in figure 19.

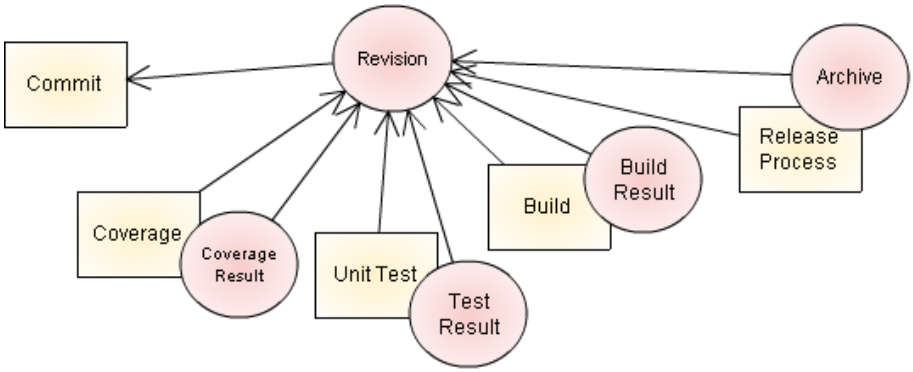

Figure 19: OPM scheme showing all processes interacting with the revision artifact.

Release On the one hand, the release artifact is composed of one or more issues. On the other hand, a release archive is derived from the release as each archive belongs to exactly one release. This is shown in figure 20.

Figure 20: OPM scheme showing all processes interacting with the release artifact.

4.4 Model

The combination of the actors, their data items and interactions leads to the final model, shown in figure 21. The individual processes are highlighted using blue clouds in the background, the interaction data items using a bold face.

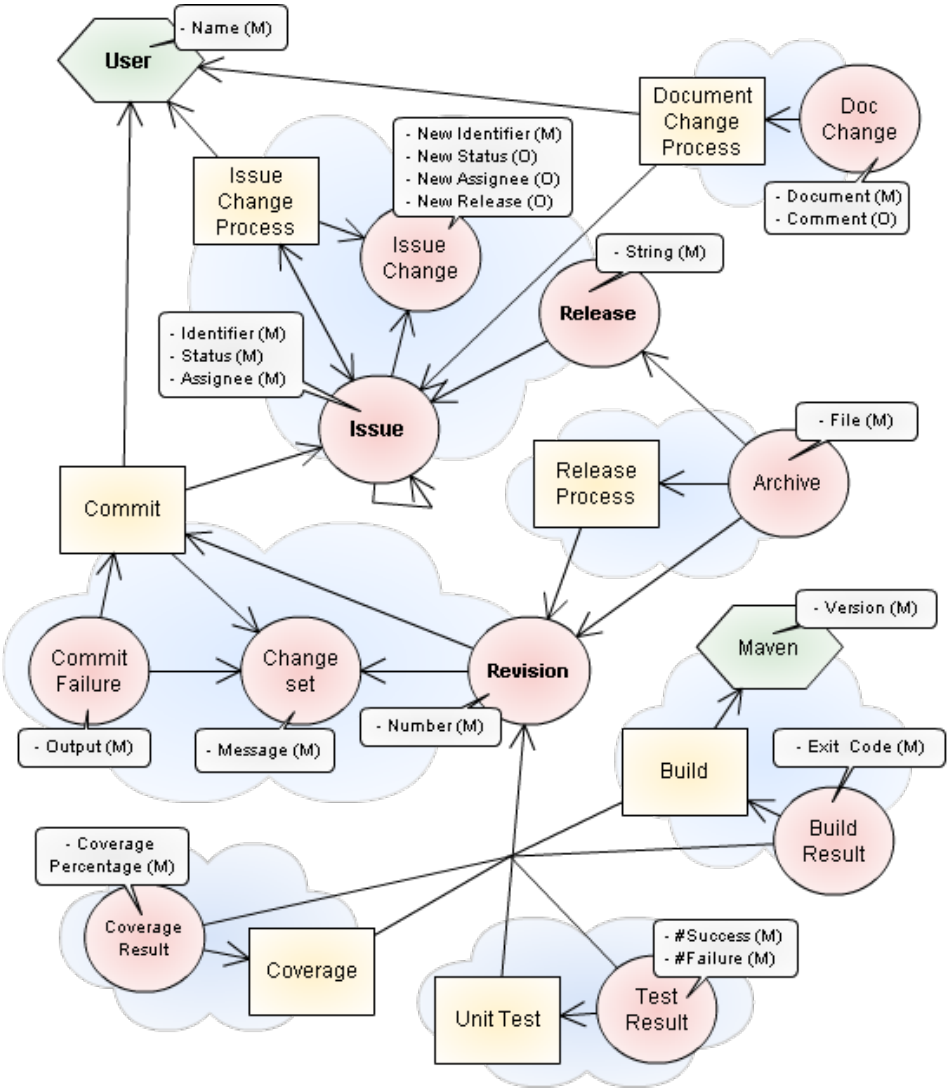

Figure 21: Complete OPM scheme for recording software development processes.

5 Implementation: Using Neo4j and Gremlin

This chapter explains how the developed Open Provenance Model scheme is implemented using the graph database Neo4j. It continues with translating the provenance questions into Gremlin queries. Finally, a complexity analysis is performed on the graph and the queries in order to determine the scalability of the solution.

5.1 Neo4j Model

Neo4j uses the property graph model, described in chapter 2.3.1, as a base for all graphs. It is now explained how the Open Provenance Model, in general and especially the developed scheme, can be mapped to this property graph model. Afterwards, a detailed example of the model filled with the data of two commits is shown.

Mapping The Open Provenance Model consists of vertices connected to each other via edges. Vertices and edges can be described in more detail using annotations. There are three different types of vertices: i) agents, ii) artifacts, iii) processes and five types of edges: i) used, ii) wasGeneratedBy, iii) wasControlledBy, iv) wasTriggeredBy, v) wasDerivedFrom.

The property graph model only supports vertices and edges, both having a unique identifier and describable by properties. Additionally, edges can have labels. The unique identifier is automatically assigned by Neo4j and cannot be altered.

The different vertice types of OPM can me modelled using properties. The different edge types of OPM are modelled using labels. OPM annotations can be mapped 1:1 to properties.

The developed scheme associates each vertex with an additional type that identifies the vertex in the context of the application, e.g., an agent is further specialized as being a user. This type is realized as a property as well.

Example Figure 22 shows an example record of two commit processes using the developed scheme, realized in a Neo4j property graph. In this example, the user wend_he (1) tries to commit (3) a changeset (4) without specifying an issue identifier to which the commit belongs. Therefore, the commit is rejected (5). In the second attempt (10) he specifies the relation to issue 1202 (2) and a new revision is created (12). The identifiers of the vertices and edges have automatically been assigned by Neo4j in order of creation.

Domain Model The translation of the scheme into a Java-based domain model, making it easier to work with the scheme, is supported by different extensions to Neo4j. These extensions automatically map the domain model to a property graph. The implementation of the prototype followed the IMDB example [neob]. Vertices are mapped to Java classes of the specialized scheme type, e.g., user. Properties are mapped to class attributes. The OPM type is encoded as a class attribute as well, relationships can automatically be derived based on these attributes.

Open Provenance Model For data exchange and compatibility purposes, the provenance community provides the OPM toolbox, containing an xsd schema, an owl ontology an a Java domain [opm]. Converting the Neo4j graph into instances of these representations can be carried out as explained in the mapping section, respecting the individual technologies. This is not explained in further detail as it is not necessary for this thesis and the feasibility has been proven in the mapping section.

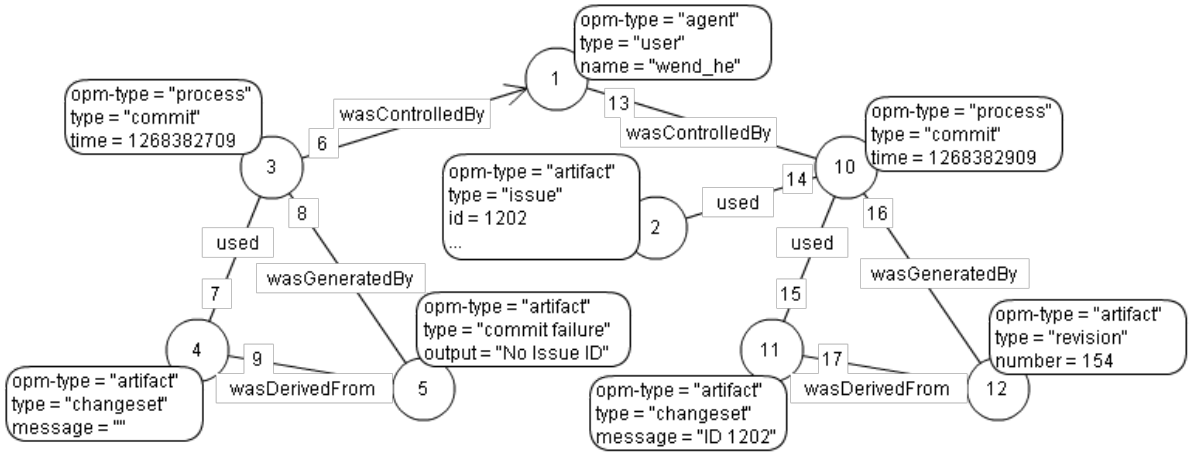

Figure 22: Example commit process implemented in neo4j.

5.2 Gremlin Queries

This section translates the provenance questions, stated in chapter 3 as human-readable-questions and analyzed using PrIMe, into Gremlin queries, which can be executed on instances of the described Neo4j model.

The given code examples do not claim to be the only and best option to answer the query, but illustrate a typical usage of Gremlin. Discussion of this topic can be found in the last chapter.

The following list explains some peculiarities of Neo4j/Gremlin:

- All values of index queries have to be specified as strings, internally the conversion to the correct type is performed automatically.
- Neo4j automatically assigns each node with a unique id in order of its creation.
- Sorting cannot be performed on nodes or vertices itself, but only on simple datatypes such as strings or integers.

1.) Who is responsible for implementing issue X? The start item of this query is the issue with the given identifier. The issue is retrieved via an index query, performed by the g:key() function (line 1). This query can return more than one issue, as each issue change generates a new issue. Therefore, the newest issue has to be selected, which is carried out by sorting the issues in descending order by their id (line 2). Afterwards, the graph is traversed along its edges to the handler of the issue (line 3).

```
1 $issues := g:key($_g, 'identifier', string($issue))
2 $id := g:sort($issues/@id, true())[1]
3 $handler := g:id($_g, $id)/outE[@label='GENERATED_BY']/inV/outE[@label='
    CONTROLLED_BY']/inV/@name
```

Listing 2: Gremlin query for provenance question 1.

2.) What is the current build status of the project? The current build status is queried by selecting all build results (line 1) and sorting them by their id (line 2). The latest build is

selected and its exit code returned (line 3).

```
1 $builds := g:key($_g, 'type', 'build_result')
2 $id := g:sort($builds/@id, true())[1]
3 $status := g:id($_g, $id)/@exit_code
```

Listing 3: Gremlin query for provenance question 2.

3.) What is the current overall code coverage? In accordance with the query above, all coverage reports are retrieved (line 1), sorted by their id (line 2) and the coverage percentage of the latest coverage report returned (line 3).

```
1 $coverages := g:key($_g, 'type', 'coverage_result')
2 $id := g:sort($coverages/@id, true())[1]
3 $percentage := g:id($_g, $id)/@percentage
```

Listing 4: Gremlin query for provenance question 3.

4.) From which revision was release X built? The start item for this query is an archive of the given release. As each release can consist of multiple archives, the first one is selected, knowing that each must be built from the same revision (line 1). Afterwards, the graph is traversed to the unique revision number (line 2).

```
1 $archive := g:key($_g, 'string', string($release))/inE[1]
2 $release := $archive/outV/outE/inV[@type='revision']/@number
```

Listing 5: Gremlin query for provenance question 4.

5.) How many unsuccessful commits did user X do? After the index search for the user (line 1) and the selection of all commits performed by him (line 2), the commits are counted using the count function (line 3).

```
1 $user := g:key($_g, 'name', string($name))
2 $commits := $user/inE/outV[@type='commit']
3 $count := count($commits)
```

Listing 6: Gremlin query for provenance question 6.

6.) How did the number of unit tests change in the last month? The most difficult part of this query is to select the unit test report from the previous month. First the current unix timestamp (line 1) and the timestamp one month ago (line 2) are calculated. Afterwards, an index query for all unit test reports is performed (line 4) and restricted to the reports between these two timestamps (line 5). This is required because it cannot be assumed that a report was generated at the exact calculated time one month ago. Finally, the results are sorted again

(line 6) and the difference between the first and last report calculated (line 8).

```
1 $now := g:time()
2 $last := g:time() - 1000*60*60*24*30
3
4 $tests := g:key($_g, 'type', 'unit_test')
5 $intime := $tests[@timestamp > $last and @timestamp < $now]
6 $ids := g:sort($intime/inE/outV/@id, true())
7
8 $change := g:id($_g, $ids[1])/@success - g:id($_g, $ids[last()])/@success
```

Listing 7: Gremlin query for provenance question 6.

7.) Which developer is most active in contributing documentation? To answer this question all documentation items have to be selected and grouped by the users that created them. All users are selected that contribute to documentation (line 1) and, afterwards, the number of documentation items for each user is saved in a map (lines 3 - 7). Finally, the developer with the highest count is selected (line 9).

```
1 $devs := g:dedup(g:key($_g, 'type', 'doc_change')/outE/inV/outE/inV[@type='user
      ']/@name)
2
3 $results := g:map()
4 foreach $dev in $devs
5    $count := count(g:key($_g, 'type', 'doc_change')/outE/inV/outE/inV[@type='user
         ' and @name=$dev])
6    g:assign($results, $dev, $count)
7 end
8
9 $best := g:keys(g:sort($results, 'value', true()))[1]
```

Listing 8: Gremlin query for provenance question 7.

8.) How many releases have been produced this year? In this query all release processes are selected (line 4), because these are the items in which the timestamp is saved. Selecting the correct range is carried out in a similar way to query 6 (line 1-2). Afterwards, the graph is traversed from the process to the actual release and duplicates that exist, as one then one archive can be created for each release, eliminated (line 5).

```
1 $now := g:time()
2 $last := g:time() - 1000*60*60*24*365
3
4 $processes := g:key($_g, 'type', 'release_process')
5 $intime := $processes[@timestamp > $last and @timestamp < $now]
6 $number := count(g:dedup($intime/inE/outV/outE/inV[@type='release']/@string))
```

Listing 9: Gremlin query for provenance question 8.

9.) How many issues were implemented by developer X for release Y? First, the given release (line 1) is selected and the graph is traversed to the issues resolved by the given developer (line 2). After removing duplicates (as the issue may have been reopened and resolved multiple times), they are counted (line 3).

```
1 $release := g:key($_g, 'string', string($release))
2 $issues := $release/outE/inV[@status='resolved' and @assignee=string($dev)]
3 $count := count(g:dedup($issues/@identifier))
```

Listing 10: Gremlin query for provenance question 9.

10.) How many commits did developer X contribute to release Y? Similar to the query above, the release is selected (line 1) and then the graph is traversed to the commits belonging to the issues associated with the release (line 2). Afterwards, the commits are limited to those handled by the given developer (line 3) and counted (line 4).

```
1 $release := g:key($_g, 'string', string($release))
2 $commits := $release/outE/inV/inE/outV[@type='commit']
3 $relevant := $commits[outE/inV[@type='user' and @name=string($developer)]]
4 $count := count($relevant)
```

Listing 11: Gremlin query for provenance question 10.

11.) How many developers contributed to issue X? The number of developers that contributed to a single issue is the union of their names with duplicates removed (line 9). The names are collected by selecting those contributing to the issue itself (lines 1,5), those committing code for the issue (lines 2,6) and those writing documentation for the issue (lines 3,7).

```
1 $issues := g:key($_g, 'identifier', string($issue))
2 $commits := $issues/inE/outV[@type='commit']
3 $docs := $issues/inE/outV[@type='document_change']
4
5 $issue_names := $issues/inE/outV[@type='issue_change_process']/outE/inV[@type='
      user']/@name
6 $commit_names := $commits/outE/inV[@type='user']/@name
7 $doc_names := $docs/outE/inV[@type='user']/@name
8
9 $number := count(g:union($issue_names, $commit_names, $doc_names))
```

Listing 12: Gremlin query for provenance question 11.

12.) Which developers contributed to release X? This query is similar to that above, except that the starting node is a release not an issue and, therefore, all relevant issues have to be selected first (line 1). Finally, the names are not counted, only collected (line 3).

```
1 $issues := g:key($_g, 'string', string($release))/outE/inV
2 ...
3 $names := g:union($issue_names, $commit_names, $doc_names)
```

Listing 13: Gremlin query for provenance question 12.

13.) How many commits were needed for issue X? After finding the issue (line 1), the commits belonging to it are selected (line 2) and counted (line 3).

```
1 $issues := g:key($_g, 'identifier', string($issue))
2 $commits := $issues/inE/outV[@type='commit']
3 $count := count($commits)
```

Listing 14: Gremlin query for provenance question 13.

14.) How much time has been spent implementing issue X? The first appearance of the issue is identified (line 1) and then the last change to the resolved status (line 2). Afterwards, the time between those two changes is calculated (lines 4-7).

```
1 $id1 := g:sort(g:key($_g, 'new_id','3291')/@id, true())[last()]
2 $id2 := g:sort(g:key($_g,'new_id','3291')[@new_status='resolved']/@id,true())[1]
3
4 $time1 := g:id($_g, $first_id)/inE/outV[@type='issue_change_process']/@timestamp
5 $time2 := g:id($_g,$last_id)/inE/outV[@type='issue_change_process']/@timestamp
6
7 $time := $time2 - $time1
```

Listing 15: Gremlin query for provenance question 14.

15.) What documentation belongs to issue X? First, the issue is selected (line 1) and then all documentation items belonging to the issue (line 2).

```
1 $issues := g:key($_g, 'identifier', string($issue))
2 $docs := $issues/inE/outV[@type='documentation_change_process']
```

Listing 16: Gremlin query for provenance question 15.

16.) Which features are part of release X? After selecting the release (line 1) the graph is traversed to all associated issues and duplicates are removed (line 2).

```
1 $release := g:key($_g,'string', string($release))
2 $issues := g:dedup($release/outE/inV/@identifier)
```

Listing 17: Gremlin query for provenance question 16.

17.) Who is responsible for reducing the code coverage? Given the two coverage reports between which the coverage was reduced, all developers who committed in between can be responsible. Therefore, first, the start revision (line 1) and second the end revision (line 2) is selected. Finally, the users responsible for the commits in between are found by traversing the graph, beginning at the given commits (line 4).

```
1 $startrev := g:id($_g, $coverages[2])/outE/inV[@type='revision']/@number
2 $endrev := g:id($_g, $coverages[1])/outE/inV[@type='revision']/@number
3
4 $names := g:key($_g, 'type', 'revision')[@number >= $startrev and @number <=
      $endrev]/outE/inV[@type='commit']/outE/inV[@type='user']/@name
```

Listing 18: Gremlin query for provenance question 17.

18.) Which requirement causes the most build failures? For each issue (line 1-4), the graph is traversed to the builds that fail after an associated commit (line 5-8) and the results are saved in a map. Finally, the highest score is returned (line 11).

```
1  $ids := g:dedup(g:key($_g, 'type', 'issue')/@identifier)
2
3  $results := g:map()
4  foreach $id in $ids
5     $issues := g:key($_g, 'identifier', string($id))
6     $revision := $issues/inE/outV[@type='commit']/inE/outV[@type='revision']
7     $build := $revision/inE/outV[@type='build']/inE/outV[@exit_code > 0]
8     g:assign($results, $id, count($build))
9  end
10
11 $most := g:keys(g:sort($results, 'value', true()))[1]
```

Listing 19: Gremlin query for provenance question 18.

19.) Which changeset resulted in more failing unit tests? Given the two unit tests reports between which the number of failing tests increased, the start revision (line 1) and the end revision (line 2) is selected. Finally, all revision numbers in between are returned (line 4).

```
1 $startrev := g:id($_g,$tests[2])/outE/inV[@type='revision']/@number
2 $endrev := g:id($_g, $tests[1])/outE/inV[@type='revision']/@number
3
4 $names := g:key($_g, 'type', 'revision')[@number >= $startrev and @number <=
      $endrev]/outE/inV[@type='commit']/outE/inV[@type='revision']/@number
```

Listing 20: Gremlin query for provenance question 19.

20.) Which version of maven caused the build failure of revision X? First, the given revision is selected (line 1) and then the graph traversed to the build executed for this revision and the corresponding maven version (line 2).

```
1 $revision := g:key($_g, 'number', string($revision))
2 $maven := $revision/inE/outV[@type='build']/outE/inV[@type='maven']/@version
```

Listing 21: Gremlin query for provenance question 20.

Example Query Visualisation Figure 23 illustrates how the graph is traversed for query 10. Beginning at the release 1.5.0, all associated issues are found and then the corresponding commits are found. Of those commits those that were performed by user wend_he are counted.

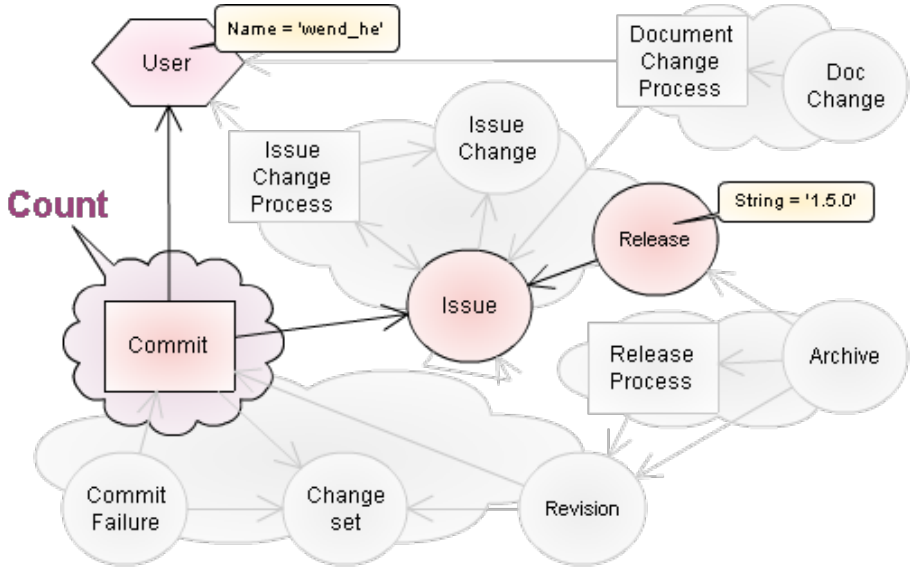

Figure 23: Visualisation of Gremlin query 10.

5.3 Complexity Analysis

This section discusses the complexity of inserting new data into the graph and executing queries on it. The complexity is not shown on all insertions and queries processes because they all show the same characteristics. Therefore, two representative examples have been selected based on the analysis of all queries.

The complexity of inserting new data into the graph is shown on the issue tracker sub-process. This process performs index queries, graph traversal and the creation and connection of new nodes with properties.

The complexity of querying the graph is shown on query 10: 'How many commits did developer X contribute to release Y?'. This query performs index queries, restriction of nodes based on properties and has a graph traversal depth of three, the maximum of all queries.

Insertion The issue tracker subprocess, is outlined in pseudo code in listing 22. An issue change is performed by a developer, changing one or more of i) the assignee, ii) the status, iii) release. An issue change is distinguished from a creation by finding if the issue already exists. First, the developer is found through an index search (line 2). All index searches are performed, assuming an appropriate indexing structure, in O(log(n)) time. N is the number of data items available for this index, in this case the number of users. If the developer does not exist he is created (lines 3-5). Creation operations have a complexity of O(1). Afterwards, the nodes for

the issue change process and the issue change are created (lines 8-9). If the issue previously existed, checked by an index query (line 11), the old values of the assignee and status are read from its property values and the release by a graph traversal of depth 1, complexity O(1) (line 12-16). These values are used as a base for the creation of the new issue (line 17). Finally, all relationships between the new nodes are created (line 26).

```
1  // find or create user
2  user = findUserByIndeSearch(name)
3  if (user == null) {
4    user = createUser(name);
5  }
6
7  // create issue change nodes
8  issueChangeProcess = createIssueChangeProcess();
9  issueChange = createIssueChange(newValues);
10
11 // find old issue and transfer values
12 oldIssue = findIssueByIndexSearch(identifier);
13 if (oldIssue != null) {
14    // Read properties and traverse graph to the release
15    newIssue = oldIssue;
16 } else {
17    newIssue = createIssue();
18 }
19 newIssue.setValues(newValues)
20
21 // create relationships
22 createRelationships(...);
```

Listing 22: Pseudo code inserting a new issue change into the graph.

In all this operation involves 3 index searches and the creation of at most 5 new nodes and 7 relationships. The biggest index is most likely that containing all issues, therefore, the upper bound of the operation is O(log(n)) where n is the number of issues in the database. The performance is logarithmically limited by the number of issues.

Query The query for all commits that a developer performed for a release, shown in listing 11, begins with an index search on the specified release. This query has a complexity of O(log(r)), where r is the number of releases (line 1). Afterwards, the graph is traversed (line 2). First, all issues associated with the release are selected . This can be accomplished by selecting all outgoing edges, as a release has only outgoing edges to issues. Afterwards, the commits (c) belonging to the issue have to be selected. Here, the incoming edges can also lead to issue changes (i) or document changes (d), therefore, a comparison of complexity O(c+i+d) has to be performed. Finally, the associated users are limited to the specified developer with a complexity of O(c). In all the complexity is O(c+i+d+log(r)) where log(r) is very small in real world cases and, therefore, can be neglected. The performance is linearly limited by the number of commits, issues and documentation items.

6 Prototype: A Service Oriented Architecture

This chapter describes how the implemented model can be integrated into the tool suite of a software development process. Using a service oriented client/server architecture, the model is automatically filled with the required data. Finally, the performance of the database containing real world data is evaluated.

6.1 Architecture

The prototype is named noblivious, derived from the verb oblivious, which means for forgetfull. The n indicates the negation, meaning nothing is forgotten. The general architecture consists of three core components:

- The Neo4j database to store all data.
- The provenance service, which implements the application specific domain model, developed in the preceding two chapters.
- The Jetty webserver [jet], which serves two of the external interfaces. For security reasons, the webserver uses HTTP BasicAuth, requiring a username and password for access.

Furthermore, three external interfaces are provided for interacting with the application:

- The REST [Fie00] interface, implemented using the Jersey [jer] library, which can be used by external tools to store new data.
- The Servlet, which can be used to query information from the database using the Gremlin query language. A static HTML page is provided displaying a console for the end user to interacts with the query interface.
- A remote interface for the Neo4j database that allows the Neoclipse plug-in to connect to remotely running instances of noblivious.

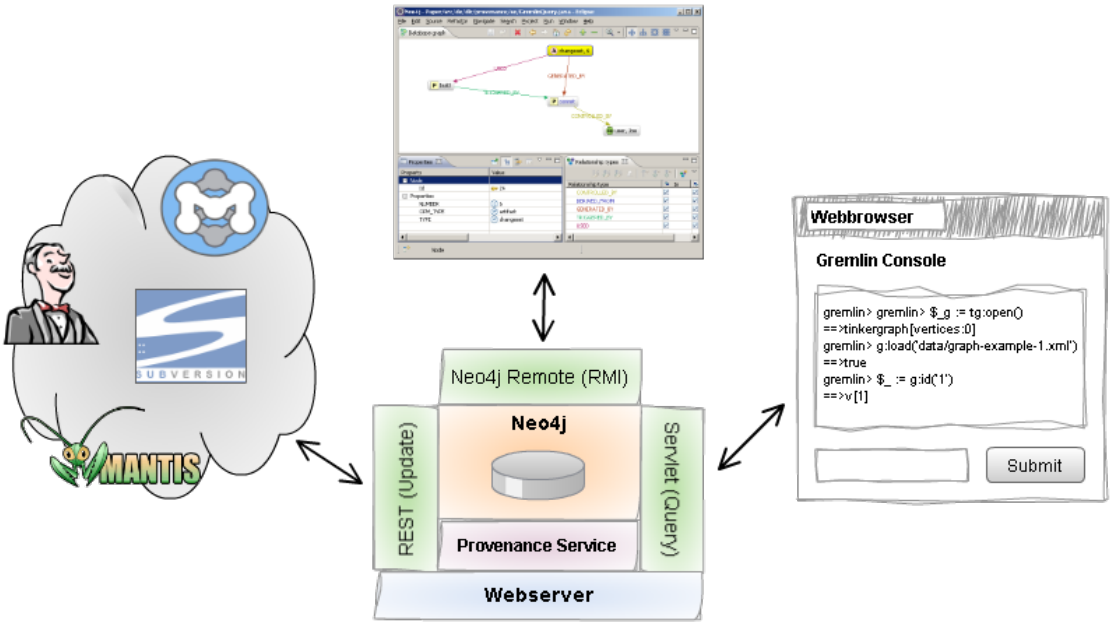

Figure 24: Architecture of the prototype software development provenance database.

6.2 REST API

Noblivious provides a REST API for external tools to insert new data. REST was chosen for its simplicity in contrast to traditional webservices. Small libraries to add REST support to applications exist for most programming languages, e.g., Jersey for Java [jer], python-rest-client [pyt] for Python, Tonic [ton] for PHP or cURL [cUR] for use on the command line.

In the following, the provided APIs are described in alphabetical order. The first row of the column states the URL to access the specific function. Afterwards, all parameters, their data types and a description are listed. Parameters are transferred using the JSON [jso] notation and HTTP POST which supports large amounts of data, such as build outputs. All methods return 'True' if the operation completed successfully or 'False' if an error occurred.

Build The build API allows to add new builds to the database. A build is specified by the revision number that was checked out, the exit code and the version of maven which executed the build.

http://host:port/rest/build		
revision	int	The revision number which was built.
result	int	The exit code of the build. 0 indicates success, everything else failure.
maven	string	The maven version used for the build.

Table 26: REST API to record builds.

Coverage The coverage API allows to add new coverage reports to the database. A coverage report is specified by the revision number it belongs to and the percentage of code that is covered by tests.

http://host:port/rest/coverage		
revision	int	The revision number to which the coverage belongs.
percentage	int	The percentage of code that is covered by tests.

Table 27: REST API to record coverage reports.

Documentation The documentation API allows to add documentation changes to the database. A documentation change is specified by the user who edited the documentation, the name of the page that was edited, the message provided while editing and the issue to which the documentation belongs.

http://host:port/rest/coverage		
user	string	The user who edited the documentation.
page	string	The name of the page that was edited.
message	string	The comment on the edit provided by the user.
issue	int	The identifier of the issue to which the change belongs.

Table 28: REST API to record documentation changes.

IssueTracker The issue tracker API allows to add issue changes to the database. An issue change is specified by the user who edits an issue, the identifier of the issue, and the change of at least one of: status, assignee or release. Issues that do not exist in the database are automatically created.

http://host:port/rest/issues		
user	string	The user who edited the issue.
id	int	The identifier of the issue.
status	string	The new status of the issue.
assignee	string	The new assignee for the issue.
release	string	The new release to which the issue is assigned.

Table 29: REST API to record issue changes.

Release The release API allows to add new releases to the database. A release is specified by the name of the release, from which it was built and the resulting file. If a release contains more than one file, the function has to be called multiple times.

http://host:port/rest/release		
release	string	The name (version number) of the new release.
revision	int	The revision from which the release was built.
file	string	The final release archive file.

Table 30: REST API to record new releases.

UnitTests The unit tests API allows to add new test reports to the database. A test report is specified by the revision to which it belongs and the number of successful and failed tests.

http://host:port/rest/tests		
revision	int	The revision number to which the test report belongs.
success	int	The number of successful unit tests.
failure	int	The number of failed unit tests.

Table 31: REST API to record unit test runs.

VCS The version control system API provides two methods: new commits and commit failures.

A new revision is specified by the user who performed the commit, the revision number, the message the user provided and the issue to which the commit belongs.

http://host:port/rest/vcs/revision		
user	string	The user who performed the commit.
number	int	The new revision number.
message	string	The message provided by the user.
issue	int	The identifier of the issue to which the commit belongs.

Table 32: REST API to record new revision.

A commit failure is specified by the user who attempted to perform the commit, the message he provided and the reason for the rejection.

http://host:port/rest/vcs/failure		
user	string	The user who performed the commit.
message	string	The message provided by the user.
output	string	The reason for the rejection given by the VCS.

Table 33: REST API to record commit failures.

6.3 Neo4j Remote

The neo4j-remote library allows to expose a Neo4j database via RMI [neod]. This makes it possible to explore remote databases using the Neoclipse plug-in the same way as local databases. During the testing phase, there was no final release of the remote support and it was not yet 100% stable, which is likely to improve in the future. Figure 25 shows the Eclipse preference page used to configure a remote database in Neoclipse as well as an active connection to the developed provenance database in the background.

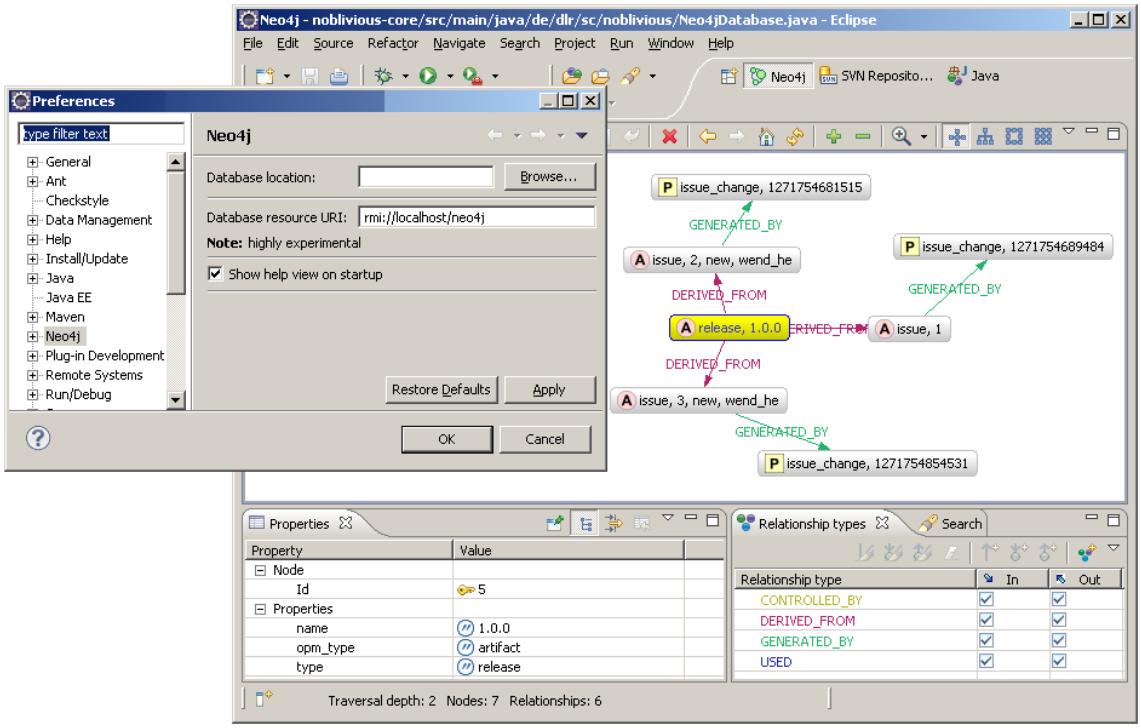

Figure 25: Neoclipse accessing the provenance database via rmi.

Listing 23 shows how to expose a remote database. It is highly recommended to install the *RMISecurityManager* (line 1), and an appropriate *java.policy* file to restrict the access to the database. A sample file is shown in figure 24, only allowing access from the server itself and the client 192.168.0.12.

```
1 System.setSecurityManager(new RMISecurityManager());
2 Registry registry = LocateRegistry.createRegistry(1099);
3 GraphDatabaseService graphDb = new EmbeddedGraphDatabase("var/graphdb");
4 RmiTransport.register(new LocalGraphDatabase(graphDb), "rmi://localhost/neo4j");
```

Listing 23: Exposing a neo4j database via RMI.

```
1 grant {
2     // Allow the server to connect and accept
3     permission java.net.SocketPermission "192.168.0.1:1024-", "accept, connect,
          resolve";
4     // Allow the client to connect and accept
5     permission java.net.SocketPermission "192.168.0.12:1024-", "accept, connect,
          resolve";
6 };
```

Listing 24: Example java.policy file to restrict access based on the IP address.

6.4 Gremlin Interface

The provenance console, shown in figure 26, provides the user with access to the database using the Gremlin query language. The console is served by a static HTML page issuing asynchronous HTTP requests to the servlet when new commands are entered. The provenance specific queries developed in this thesis are integrated as custom Gremlin functions. An overview of the commands and their usage is provided by the provenance:help() command. The figure shows the execution of provenance query 10: 'How many commits did developer wend_he contribute to release 1.5.0?'. The answer is 100.

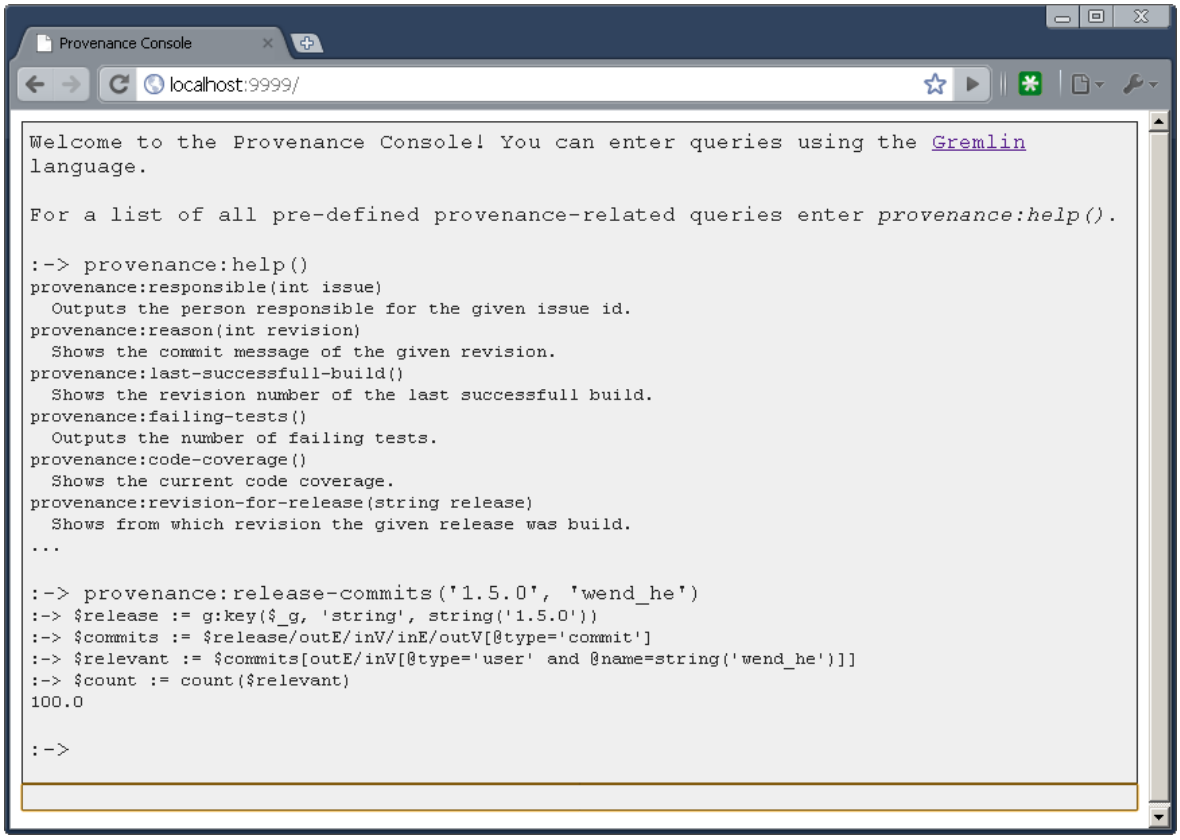

Figure 26: The provenance web console accepting Gremlin queries.

6.5 Integration into the Process

The automatic integration into the software development process is one of the essential aspects of the proposed approach. The database is useless without containing data and nobody manually enters the data in practice. Therefore, REST API is provided for tools used in the process, so that they can automatically record the users actions into the provenance database. The calls to this API have to be made at the correct moment, e.g., when the user enters a new issue, performs a commit, changes a page or the continuous integration system finished a build. Most of tools used today support some form of hook concept. These hooks serve two purposes:

- Validate the action of a user. E.g., the user has to enter an issue identifier in the commit message otherwise the commit is rejected.
- Perform additional operations on the occurrence of a specified event, such as logging or a call to the provenance REST API.

The following paragraphs outline the hook concepts of the tools used for the RCE development.

Mantis The hook concept of Mantis is named custom functions. By implementing these functions in the file named *custom_functions_inc.php*, they are executed on occurrence of their event. The important functions for this application are the notifications about the creation and the update of issues. It is also possible to validate a change before it is performed, but this is not necessary. Listing 25 shows the signatures of these methods.

```
1 function custom_function_issue_create_validate( $p_new_bug );
2 function custom_function_issue_update_notify($p_issue_id);
3
4 function custom_function_issue_create_validate( $p_new_bug );
5 function custom_function_issue_update_validate($p_issue_id, $p_new_bug,
      $p_bug_note_text);
```

Listing 25: Mantis custom functions to monitor issue changes.

Repoguard / Subversion Repoguard provides a framework to easily write hook scripts for different version control systems, subversion amongst others. Subversion supports the execution of hook scripts before a commit (pre-commit) for validation purposes and after a commit (post-commit) for logging. Repoguard can be extended through two concepts: checks and handlers. Based on the incoming transaction, a check validates the commit. The output of all checks is then forwarded to the handlers which, e.g., send emails or update an RSS feed. Repoguard is configurable in a very flexible way, allowing for the execution of different checks and handlers in pre-commit and post-commit phases.

The recording of new commits or commit failures can be realized as handler. Listing 26 shows the basic structure of this handler. It subclasses the *Handler* class and implements the *summarize()* method, which is automatically called by the framework. In this method the *transaction* can be used to extract all necessary information about new files, the revision number and commit message, and the REST API can be called.

```
1 class Provenance(Handler):
2     def _summarize(self, config, protocol):
3         recordProvenance(self.transaction)
```

Listing 26: Basic structure of a Repoguard handler.

MoinMoin The MoinMoin Wiki engine provides an event system. Different events are supported, the most important being the *PageChangedEvent*, which is also called when new pages are created. On occurrence of the event, all registered event handlers (*cfg.event_handlers*) are called. Listing 27 shows an example of such an event handler. The *event* object contains all information about the change, e.g., which page was changed or the provided message.

```
1 import MoinMoin.events as ev
2 def handle(event):
3     if isinstance(event, ev.PageChangedEvent):
4                 recordProvenance(event)
```

Listing 27: The MoinMoin event system.

Hudson The continuous integration system Hudson supports an extension point concept. One of these extension points is named notifier and executed after a build has finished. By subclassing the class *Notifier*, implementing the method *perform*, shown in listing 28, access to all information collected during the build is available via the *build* object. The registration of the extension is automatically performed by Hudson after installation of the notifier.

```
1 class Provenance(hudson.tasks.Notifier) {
2         public boolean perform(AbstractBuild<?, ?> build, Launcher launcher,
              BuildListener listener) throws InterruptedException, IOException {
3             recordProvenance(build);
4     }
5 }
```

Listing 28: The Hudson notifier interface.

Release Script Releases are created automatically in RCE with the help of a bash script. This bash script can be extended using a command line REST client to automatically record the required provenance information.

6.6 Performance Evaluation

The Neo4j homepage states: "Neo4j can handle graphs of several billion nodes/relationships/properties on a single machine". The performance of the prototype was evaluated by importing the data from the various tools used for RCE development. Table 34 provides a statistical overview of the data produced during the 5 years development. After importing this data into the prototype via the REST interface the database holds 114102 nodes, 177261 relationships and 364931 property values.

Project Runtime	5 years
Contributers	44 (17 Developers)
Issues / Edits	1034 / 4914
Wiki Pages / Edits	589 / 4155
Commits	10107
Releases	15 + 42 Snapshots

Table 34: Statistics of the RCE Project.

The performance was measured for the two algorithms discussed in the complexity analysis in chapter 5.3. Three different measurements were performed, the first when the database is still empty, the second with half of the data entered and the third with the complete set of data imported. To eliminate outliers, each measurement was performed ten times. The average of the ten measurements and the first value is listed, as this value always differed from the others. Table 35 summarizes the results.

Both the insertion and the query, involve index searches. The outlying first value is most likely explained by loading the index from the disk into main memory, where it can be accessed afterwards. Therefore, the first value is ignored in the following discussion.

The insertion consists of a static part, creation of nodes, properties and relationships, as well as a dynamic part, index queries for nodes to which the new elements can be connected to. The value of the empty database shows the performance of the static part, as no elements can be returned by the index query. The additional time needed to insert data into a half and a full database is, therefore, completely allocated to the index query, showing an expected logarithmic behaviour.

The query consists of two parts: index search and graph traversals. The empty database returns no results in the index search, as no graph has to be traversed, resulting in a time of 0ms. The increase of time in the half and then the full database shows a linear behaviour. This indicates that the logarithmic part of the index search, identified in the complexity analysis, is very small and the majority of the time is spent traversing the graph. Therefore, doubling the data size results in twice as much time needed for the query.

	Insertion		**Query**	
	First	Avg. 10	First	Avg. 10
Empty Database	782ms	303ms	47ms	0ms
Half Database	1047ms	345ms	1110ms	204ms
Full Database	1297ms	350ms	3890ms	415ms

Table 35: Performance evaluation of the prototype.

Altogether the absolute values are in a range that allows the algorithms to be useable in day to day use. For larger projects the query algorithm may need to be optimized in the linear part.

7 Conclusions: To be continued...

The content of this master thesis are summarized in the poster shown in picture 27. The software development process was analyzed (top right) and typical questions, arising on a daily basis, collected and categorized (top left). Afterwards, a provenance model was created using PrIMe and the OPM (bottom left), which can hold all data necessary to answer the questions. Finally, the model was implemented using a service oriented architecture to integrate it into the distributed tool suite and the graph database Neo4j to store the data. This makes it possible to answer the questions using the graph programming language Gremlin (bottom right).

The thesis contributed three new points to the research area of provenance:

- A new use case for collecting provenance information was shown, namely software development processes.
- PrIMe was adapted to not only handle applications, but also processes and use the Open Provenance Model.
- The data was stored using the graph database Neo4j and queried using the graph programming language Gremlin.

The basic conditions of this approach were:

- Not all possible questions can be answered by the developed model. The developed model is specially designed for the questions stated. Still, the approach of the thesis, using PrIMe to analyze processes, can be applied to other questions as well.
- The developed approach does only work if the required data and their relations are recorded automatically, e.g., to which issue a commit belongs. This can be assured by the development process and the tools used.

Still, some rough edges, points to consider and ideas for the future remain, which are summarized in the following paragraphs.

Modelling While the described graph model worked quite well for the given use case, it can still be optimized. The Neo4j best practices guide [neoa] suggests that i) Use reference and subreference nodes to organize entry points and ii) Keep your graph connected. While the first suggestion, connecting all nodes to the root node as reference, is not needed, if you use indexes as entry points, the second suggestion can be discussed. The current modelling approach already focuses on keeping the graph connected, which helps to visually navigate the graph. The downside is that it requires index operations during the insertion of new data. It would be interesting to compare the performance if these index operations are not required.

An additional way to improve the performance is to introduce concrete relationship types between different node types, not relying on the provenance types only. This reduces the number of comparison needed while traversing the graph, as described in chapter 5.3 and 6.6. It is also questionable if the OPM is needed for modelling the process at all, or if arbitrary graphs could be used. The OPM lacks some relationships, e.g., between agent and artifact, and introduces some overhead, e.g., the only data currently stored in the process nodes is the timestamp, which often increases the query size.

While the graph structure is very well suited for storing the relationships between the different tools and their artifacts, it is uncertain if saving the actual data in properties is the best option.

First, this results in replication of the data that is already available in the tools itself. Second, this example focused on only a small part of the data, e.g., an issue in this case had only three properties, while a real issue has at least 27 properties when using Mantis. If the actual data is not saved in the model, the next challenge would be how to query it. Currently Gremlin can only access the graph structure, but is extensible via custom functions.

Finally, the different options of saving changing data can be compared against one another. The examples in this model are the issue tracking and the documentation processes. While the issue tracking process saves all change requests and the resulting new issues, the documentation process only changes the save requests. Depending on the questions it is also imaginable that only the final document is saved. These options have to be evaluated in more detail.

Indexing Index structures provide the best possibility to improve the performance for queries. Unfortunately, the indexing capabilities of Neo4j/Gremlin are not handled as first class citizens, they are only made up as an extension of the actual core. This can quickly be seen when working with the tools, e.g., the standard interface only offers the possibility to query for one index key matching an exact value. These limitations are unnecessary because the powerful lucene index engine is used in the background, featuring a complete query grammar [lucb]. Although lucene primary handles string data, it can be abused to correctly handle numeric values as well. For time queries, it has to be investigated how to correctly use the timeline feature of Neo4j [neof]. The aim must be to improve the integration of these index capabilities into Neo4j/Gremlin.

Gremlin In contrast to query languages such as SQL, Gremlin is a graph-based programming language. Although it is turing complete and all queries can be expressed with it, it lacks the simplicity of specifying queries the SQL way. This can, for example, be seen when attempting to sort nodes by a given property or the missing group by statement. As Gremlin is extensible via custom functions, some work should be investigated in adding functions in order to make working with Gremlin more easy.

Queries The process of translating provenance questions into textual Gremlin queries, although being quite handy after some practice for technically experienced people, can be optimized to be useable for non-technical people, e.g., managers, as well. In any case, it requires knowledge of the meta model of the concrete graph. Neo4j currently does not have support for real meta models, they must be implemented as custom domain model including concepts such as validation. The Neo4j team works on introducing a meta model concept for the 1.1 release [neoc]. Such a meta-model could be used to provide an auto-completion for Gremlin queries in the provenance console.

Visualisation Furthermore, a meta-model could be used together with a graphical representation such as Neoclipse, to visually define queries on the graph, and provide a query mechanism that is useable by everybody. Different visualisations can be provided for a provenance graph, showing only important parts of the thousands of nodes and edges in the graph. A timeline, e.g., could show the order of sub-processes, helping to make the whole process traceable.

Data Analysis The questions building the base for analysing the software development process in this thesis were collected during a survey of developers. Although this was sufficient for showing how provenance technologies can be applied, they did not cover every possible use case. The research area of mining of software repositories mainly focuses on individual repositories currently, but some projects already try to connect the data of different repositories, such as

Alitheia [GKS08]. It would be interesting to see how these questions can be answered using a graph based repository, which already connects data between different tools.

Reasoning This thesis focused on stating questions and creating an appropriate model that contains all data required to answer these questions. A different approach would be to to take the developed model and try to categorize all possible questions that are answerable. The queries in this thesis have been manually specified using Gremlin queries, another approach would be to attempt to answer question using technologies from the research area of automatic reasoning.

Process Validation The development of medical software points out another interesting problem: process validation. Medical software must be developed following a certain software development process. This is not only true for medical software, as some customers may also requires certain processes. The provenance model could be used to verify the compliance to such processes.

Process Optimization Extended by some form of warning system, errors, regardless of whether related to the process itself or the contents of the process, could be identified in each phase of the development and relevant people notified. Based on this information bottle necks in the process or, e.g., the tools used for development can be identified and optimized.

Adaption to other Processes The focus of this thesis was the handling of questions arising in software development processes. Although software development processes are by their nature supported by a wealth of software, this is also true for other development processes. Development without the help of software is no longer possible in, e.g., the systems design or engineering domains, and even business processes, not related to development at all, are modelled using appropriate tools. The presented modelling approach, the usage of graph databases and the implementation, using a service oriented architecture, could be transferred to these processes as well, helping to find problems, validate and optimize them.

Privacy Last but not least some points besides the technical issues have to be considered. Dependent on the effective laws of the country, the company or its work council it must first be verified that recording of the process data is permitted at all. The analysis of the data may only be allowed for specified purposes, such as validation of the process, not for a rating of employees. Access may only be granted to persons in special roles, or restricted based on the positions of the individual persons. All of this has to be clarified during the planning phase already, otherwise a rude awaking may come after the implementation, during the rollout, wasting a lot of money.

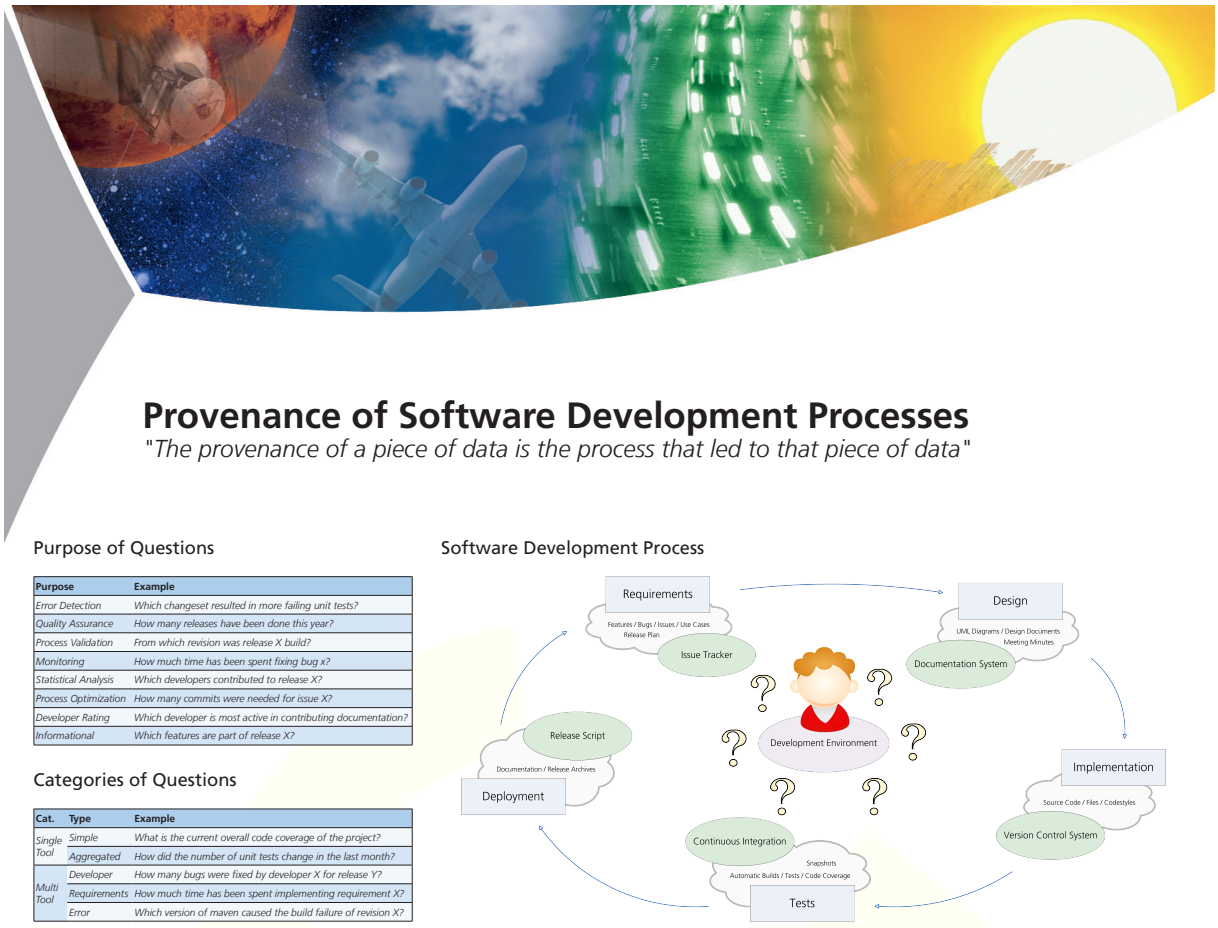

Figure 27: The big picture.

References

[BBvB+01] Beck, Kent, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, and Dave Thomas: *Manifesto for Agile Software Development*. `http://agilemanifesto.org/.org/`, 2001. Retrieved: 9th of June 2010.

[BHS09] Bock, Michael, Anita Hermann, and Tobias Schlauch: *DLR Software Projekt- und Entwicklerhandbuch*. Technical report, German Aerospace Center, Insitute Simulation and Softwaretechnologie, 2009.

[BKcT01] Buneman, Peter, Sanjeev Khanna, and Wang chiew Tan: *Why and Where: A Characterization of Data Provenance*. In *In ICDT*, pages 316–330. Springer, 2001.

[BL03] Barry, C. and M. Lang: *A comparison of 'traditional' and multimedia information systems development practices*. Information and Software Technology, 45(4):217 – 227, 2003.

[cha] *Third Provenance Challenge*. `http://twiki.ipaw.info/bin/view/Challenge/ThirdProvenanceChallenge`. Retrieved: 9th of June 2010.

[che] *Checkstyle*. `http://checkstyle.sourceforge.net/`. Retrieved: 9th of June 2010.

[cob] *Cobertura*. `http://cobertura.sourceforge.net/`. Retrieved: 9th of June 2010.

[Coc01] Cockburn, Alistair: *Agile Software Development.: Software Through People*. Addison-Wesley Longman, 2001.

[cUR] *cURL and libcurl*. `http://curl.haxx.se/`. Retrieved: 9th of June 2010.

[cvs] *CVS - Concurrent Versions System*. `http://www.nongnu.org/cvs/`. Retrieved: 9th of June 2010.

[Dav90] Davis, Alan M.: *Software requirements: analysis and specification*. Prentice Hall Press, Upper Saddle River, NJ, USA, 1990.

[DLFOT06] De Lucia, Andrea, Fausto Fasano, Rocco Oliveto, and Genoveffa Tortora: *Can Information Retrieval Techniques Effectively Support Traceability Link Recovery?* In *ICPC '06: Proceedings of the 14th IEEE International Conference on Program Comprehension*, pages 307–316, Washington, DC, USA, 2006. IEEE Computer Society.

[dlra] *Distributed Systems and Component Software*. `http://www.dlr.de/sc/en/desktopdefault.aspx/tabid-1199/1657_read-3066/`. Retrieved: 9th of June 2010.

[dlrb] *DLR at a glance*. `http://www.dlr.de/en/desktopdefault.aspx/tabid-636/1065_read-1465/`. Retrieved: 9th of June 2010.

[dlrc] *Simulation and Software Technology*. `http://www.dlr.de/sc/en/desktopdefault.aspx/tabid-1185/1634_read-3062/`. Retrieved: 9th of June 2010.

[DPM00] Dutoit, Allen, Barbara Paech, and Technische Universität München: *Rationale Management in Software Engineering*, 2000.

[dVGT09] Virgilio, Roberto de, Fausto Giunchiglia, and Letizia Tanca: *Semantic Web Information Management: A Model-Based Perspective*. Springer, 2009.

[Dvo01] Dvorak, Daniel L.: *NASA Study on Flight Software Complexity*. Technical report, NASA Office of Chief Engineer, 2001.

[ecl] *Explore the Eclipse universe...* `http://eclipse.org/`. Retrieved: 9th of June 2010.

[Eif09] Eifrem, Emil: *Neo4j - the benefits of graph databases*. In *no:sql(east)*, 2009.

[EN03] Elmasri, Ramez A. and Shankrant B. Navathe: *Fundamentals of Database Systems*. Addison-Wesley Longman Publishing Co., Inc., 2003.

[Fie00] Fielding, Roy Thomas: *Architectural styles and the design of network-based software architectures*. PhD thesis, 2000. Chair-Taylor, Richard N.

[GF94] Gotel, Orlena C. Z. and Anthony C. W. Finkelstein: *An Analysis of the Requirements Traceability Problem*. In *Proceedings of the First International Conference on Requirements Engineering*, pages 94–101, 1994.

[git] *Git - Fast Version Control System*. `http://git-scm.com/`. Retrieved: 9th of June 2010.

[GJM+06] Groth, Paul, Sheng Jiang, Simon Miles, Steve Munroe, Victor Tan, Sofia Tsasakou, and Luc Moreau: *An architecture for provenance systems*. Technical report, University of Southampton, November 2006.

[GKS08] Gousios, Georgios, Eirini Kalliamvakou, and Diomidis Spinellis: *Measuring developer contribution from software repository data*. In *MSR '08: Proceedings of the 2008 international working conference on Mining software repositories*, pages 129–132, New York, NY, USA, 2008. ACM.

[GMT+06] Groth, P., S. Munroe, V. Tan, Sheng Jiang, S. Miles, and L. Moreau: *The P-assertion Recording Protocol*. Technical report, 2006.

[gre] *Gremlin - a graph-based programming language*. `http://wiki.github.com/tinkerpop/gremlin/`. Retrieved: 9th of June 2010.

[Has08] Hassan, A.E.: *The road ahead for Mining Software Repositories*. In *Frontiers of Software Maintenance, 2008. FoSM 2008.*, pages 48–57, October 2008.

[HBM+08] Holland, D. A., U. Braun, D. Maclean, K. K. Muniswamy-Reddy, and M. Seltzer: *Choosing a Data Model and Query Language for Provenance*. In *Second International Provenance and Annotation Workshop (IPAW'08)*, 2008.

[HGH08] Hindle, Abram, Daniel M. German, and Ric Holt: *What do large commits tell us?: a taxonomical study of large commits*. In *MSR '08: Proceedings of the 2008 international working conference on Mining software repositories*, pages 99–108, New York, NY, USA, 2008. ACM.

[hud] *Hudson - Extensible continuous integration server*. `https://hudson.dev.java.net/`. Retrieved: 9th of June 2010.

[HWR+07] Hunter, David, Andrew Watt, Jeff Rafter, Jon Duckett, Danny Ayers, Nicholas Chase, Joe Fawcett, Tom Gaven, and Bill Patterson: *Beginning XML*. Wiley Publishing , Inc., 2007.

[jen] *Jena A Semantic Web Framework for Java*. http://jena.sourceforge.net/. Retrieved: 9th of June 2010.

[jer] *Jersey: the open source JAX-RS (JSR 311) Reference Implementation for building RESTful Web services*. https://jersey.dev.java.net/. Retrieved: 9th of June 2010.

[jet] *jetty - Jetty WebServer*. http://jetty.codehaus.org/jetty/. Retrieved: 9th of June 2010.

[jso] *JSON (JavaScript Object Notation)*. http://www.json.org/. Retrieved: 9th of June 2010.

[jun] *JUnit.org Resources for Test Driven Development*. http://www.junit.org/. Retrieved: 9th of June 2010.

[Ker09] Kerievsky, Joshua: *So You Want To Be A Programming Rock Star? - Google Tech Talk*. http://www.youtube.com/watch?v=XVfJSqAhHV8, 2009. Retrieved: 9th of June 2010.

[Kru03] Kruchten, Philippe: *The Rational Unified Process: An Introduction*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[KS09] Kannenberg, Andrew and Dr. Hossein Saiedian: *Why Software Requirements Traceability Remains a Challenge*. CrossTalk The Journal of Defense Software Engineering, July 2009.

[KZPW06] Kim, Sunghun, Thomas Zimmermann, Kai Pan, and E. James Jr. Whitehead: *Automatic Identification of Bug-Introducing Changes*. In *ASE '06: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, pages 81–90, Washington, DC, USA, 2006. IEEE Computer Society.

[LML09] Lungu, Mircea, Jacopo Malnati, and Michele Lanza: *Visualizing Gnome with the Small Project Observatory*. Mining Software Repositories, International Workshop on, 0:103–106, 2009.

[LPR+09] Legenhausen, Malte, Stefan Pielicke, Jens Ruhmkorf, Heinrich Wendel, and Andreas Schreiber: *RepoGuard: A Framework for Integration of Development Tools with Source Code Repositories*. International Conference on Global Software Engineering, pages 328–331, 2009.

[luca] *Lucene*. http://lucene.apache.org/. Retrieved: 9th of June 2010.

[lucb] *Lucene - Query Parser Syntax*. http://lucene.apache.org/java/1_4_3/queryparsersyntax.html. Retrieved: 9th of June 2010.

[man] *Mantis Bug Tracker*. http://www.mantisbt.org/. Retrieved: 9th of June 2010.

[mav] *Welcome to Apache Maven*. http://maven.apache.org/. Retrieved: 9th of June 2010.

[MCF+09] Moreau, Luc, Ben Clifford, Juliana Freire, Yolanda Gil, Paul Groth, Joe Futrelle, Natalia Kwasnikowska, Simon Miles, Paolo Missier, Jim Myers, Yogesh Simmhan, Eric Stephan, and Jan Van den Bussche: *The Open Provenance Model Core Specification 1.1*, 2009.

[MFG+09] Myers, James D., Joe Futrelle, Jeff Gaynor, Joel Plutchak, Peter Bajcsy, Jason

Kastner, Kailash Kotwani, Jong Sung Lee, Luigi Marini, Rob Kooper, Robert E. McGrath, Terry McLaren, Alejandro Rodríguez, and Yong Liu: *Embedding Data within Knowledge Spaces*. CoRR, abs/0902.0744, 2009.

[MGM+08] Moreau, Luc, Paul Groth, Simon Miles, Javier Vazquez-Salceda, John Ibbotson, Sheng Jiang, Steve Munroe, Omer Rana, Andreas Schreiber, Victor Tan, and Laszlo Varga: *The provenance of electronic data*. Commun. ACM, 51(4):52–58, 2008.

[MLA+08] Moreau, Luc, Bertram Ludäscher, Ilkay Altintas, Roger S. Barga, Shawn Bowers, Steven P. Callahan, George Chin Jr., Ben Clifford, Shirley Cohen, Sarah Cohen Boulakia, Susan B. Davidson, Ewa Deelman, Luciano A. Digiampietri, Ian T. Foster, Juliana Freire, James Frew, Joe Futrelle, Tara Gibson, Yolanda Gil, Carole A. Goble, Jennifer Golbeck, Paul T. Groth, David A. Holland, Sheng Jiang, Jihie Kim, David Koop, Ales Krenek, Timothy M. McPhillips, Gaurang Mehta, Simon Miles, Dominic Metzger, Steve Munroe, Jim Myers, Beth Plale, Norbert Podhorszki, Varun Ratnakar, Emanuele Santos, Carlos Eduardo Scheidegger, Karen Schuchardt, Margo I. Seltzer, Yogesh L. Simmhan, Cláudio T. Silva, Peter Slaughter, Eric G. Stephan, Robert Stevens, Daniele Turi, Huy T. Vo, Michael Wilde, Jun Zhao, and Yong Zhao: *Special Issue: The First Provenance Challenge*. Concurrency and Computation: Practice and Experience, 20(5):409–418, 2008.

[MMG+06] Munroe, S., S. Miles, P. Groth, S. Jiang, V. Tan, L. Moreau, J. Ibbotson, and J. Vazquez-Salceda: *PrIMe: A Methodology for Developing Provenance-Aware Applications*, 2006.

[moi] *The MoinMoin Wiki Engine*. `http://moinmo.in/`. Retrieved: 9th of June 2010.

[Mor09] Moreau, Luc: *The Foundations for Provenance on the Web*. Technical report, University of Southampton, 2009.

[neoa] *Neo4j - Guidelines for Building a Neo4j Application*. `http://wiki.neo4j.org/content/Guidelines_for_Building_a_Neo4j_Application`. Retrieved: 9th of June 2010.

[neob] *Neo4j - IMDB Domain Services*. `http://wiki.neo4j.org/content/IMDB_Domain_Services/`. Retrieved: 9th of June 2010.

[neoc] *Neo4j - Meta Model*. `http://components.neo4j.org/neo4j-meta-model/`. Retrieved: 9th of June 2010.

[neod] *Neo4j - Remote Graph Database*. `http://components.neo4j.org/neo4j-remote-graphdb/`. Retrieved: 9th of June 2010.

[neoe] *Neo4j - the graph database*. `http://www.neo4j.org/`. Retrieved: 9th of June 2010.

[neof] *Neo4j indexing*. `http://components.neo4j.org/neo4j-index/`. Retrieved: 9th of June 2010.

[neog] *Neoclipse Guide*. `http://wiki.neo4j.org/content/Neoclipse_Guide`. Retrieved: 9th of June 2010.

[nex] *Nexus — the repository manager*. `http://nexus.sonatype.org/`. Retrieved: 9th of June 2010.

[NR68] Naur, Peter and Brian Randell (editors): *Nato Software Engineering Conference*, 1968.

[opm] *Open Provenance Model (OPM).* `http://openprovenance.org/`. Retrieved: 9th of June 2010.

[pyt] *python-rest-client: A REST client providing a convenience wrapper over httplib2.* `http://code.google.com/p/python-rest-client/`. Retrieved: 9th of June 2010.

[rce] *Remote Component Environment.* `http://www.rcenvironment.de/`. Retrieved: 9th of June 2010.

[Roy87] Royce, W. W.: *Managing the development of large software systems: concepts and techniques.* In *ICSE '87: Proceedings of the 9th international conference on Software Engineering*, pages 328–338, Los Alamitos, CA, USA, 1987. IEEE Computer Society Press.

[Sch06] Schwaber, Carey: *The Changing Face Of Application Life-Cycle Management.* Technical report, Forrester, 2006.

[Sch07] Schreiber, Andreas: *Python in der Luft- und Raumfahrt.* In *Python im deutschsprachigen Raum, Python Academy, SSchwarzer.com und Universität Leipzig*, 2007.

[ses] *openRDF.org ... home of Sesame.* `http://www.openrdf.org/`. Retrieved: 9th of June 2010.

[Som07] Sommerville, Ian: *Software Engineering*, chapter 4. Addison Wesley, 2007.

[SPG05] Simmhan, Yogesh L., Beth Plale, and Dennis Gannon: *A survey of data provenance in e-science.* SIGMOD Rec., 34(3):31–36, 2005.

[svn] *Apache Subversion.* `http://subversion.apache.org/`. Retrieved: 9th of June 2010.

[SWS09] Seider, Doreen, Joachim Wend, and Andreas Schreiber: *Embedding Existing Heterogeneous Monitoring Techniques into a Lightweight, Distributed Integration Platform.* In *D-Grid Monitoring Workshop*, 2009.

[TC09] Tylissanakis, Giorgos and Yiannis Cotronis: *Data Provenance and Reproducibility in Grid Based Scientific Workflows.* In *GPC '09: Proceedings of the 2009 Workshops at the Grid and Pervasive Computing Conference*, pages 42–49. IEEE Computer Society, 2009.

[ton] *Tonic: RESTful Web App Development PHP Library.* `http://tonic.sourceforge.net/`. Retrieved: 9th of June 2010.

[Tro07] Troelsen, Andrew: *Pro C# 20008 and the .NET 3.5 Platform.* apress, 2007.

[udc] *Usage Data Collector.* `http://www.eclipse.org/epp/usagedata/`. Retrieved: 9th of June 2010.