Institute of Software Technology
Reliable Software Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Master's Thesis

Issue Management for Multi-Project, Multi-Team Microservice Architectures

Sandro Speth

Course of Study:	Softwaretechnik
Examiner:	Prof. Dr.-Ing. Steffen Becker
Supervisor:	Prof. Dr.-Ing. Steffen Becker, Dr. rer. nat. Uwe Breitenbücher
Commenced:	May 29, 2019
Completed:	November 25, 2019
CR-Classification:	D.2.2

Abstract

Many modern software architectures follow a microservice style. An microservice architecture consists of several independent developed and operated services. Often, issues, e.g. interface model changes, or design decision changes must be communicated between multiple teams. For example, if there are any changes of an interface of a service, all depending services must be changed too, otherwise their functionality might break. Changes in artefacts and models concerning multiple teams, therefore, must be synchronized between all affected teams in order to be in a consistent state. However, this is difficult and current approaches to communicate issues affecting multiple projects or teams comes with a communication overhead. Various methods are used to communicate them, for example, e-mail, instant messengers, calls or additional meetings. In order to synchronize models or design decisions, traceability towards them can be used as suggested by some researches. Yet, this does not solve the communication problem. This thesis introduces multi-project coding issues as solution approach for communicating issues concerning multiple projects/services and teams in a qualitative way. A multi-project coding issue is a coding issue that can concern more than one projects at the same time. They can link other coding issues which could concern other projects/services. Since their body text can be extended with semantic links, multi-project coding issues build a perfect platform to include such traceability links mentioned. Therefore, artefacts can easily be synchronized over multiple teams. This reduces development complexity while keeping communication overhead small. In addition to this, multi-project coding issues can have non-functional requirements to improve quality-of-service properties. This work presents the Multi-Project Issue Management and Notation, a modelling language to notate multi-project coding issues and projects/services together in a system architecture graph. Furthermore, a prototype of a multi-project coding issue management system is described. To validate this solution approach a Goal-Question-Metric plan is depicted, and an expert survey conducted. Finally, future research challenges are introduced.

Kurzfassung

Viele moderne Softwarearchitekturen folgen einem Microservice-Stil. Eine Microservice-Architektur besteht aus mehreren unabhängig entwickelten und betriebenen Diensten. Häufig müssen Probleme, wie z.B. Änderungen an Schnittstellenmodellen oder Änderungen an Designentscheidungen, zwischen mehreren Teams kommuniziert werden. Wenn es beispielsweise Änderungen an einer Schnittstelle eines Dienstes gibt, müssen auch alle abhängigen Dienste angepasst werden, da sonst deren Funktionalität beeinträchtigt werden könnte. Änderungen an Artefakten und Modellen, die mehrere Teams betreffen, müssen daher zwischen allen betroffenen Teams synchronisiert werden, um in einem konsistenten Zustand zu sein. Dies ist jedoch schwierig und die aktuellen Ansätze zur Kommunikation von Problemen (Issues), die mehrere Projekte oder Teams betreffen, sind mit einem Kommunikationsaufwand verbunden. Zur Kommunikation werden verschiedene Methoden eingesetzt, z.B. E-Mail, Instant Messenger, Telefonkonferenzen oder zusätzliche Meetings. Um Modelle oder Designentscheidungen zu synchronisieren, können (Verfolgbarkeits-)Links zu ihnen genutzt werden, wie von einigen wissenschaftlichen Ausarbeitungen vorgeschlagen wird. Dies löst jedoch nicht das Kommunikationsproblem. Diese Arbeit stellt Multi-Project Coding Issues als Lösungsansatz für die Kommunikation von Problemen, die mehrere Projekte/Dienstleistungen und Teams betreffen, auf qualitative Weise vor. Ein Multi-Project Coding Issue ist ein Coding Issue, das mehr als ein Projekt gleichzeitig betreffen kann. Sie können andere Coding Issues verknüpfen, die andere Projekte/Dienste betreffen könnten. Da ihr Textkörper mit semantischen Links erweitert werden kann, bilden Multi-Project Coding Issues eine perfekte Plattform, um die genannten (Verfolgbarkeits-)Links aufzunehmen. Somit können Artefakte problemlos über mehrere Teams hinweg synchronisiert werden. Dies reduziert die Komplexität der Entwicklung und hält den Kommunikationsaufwand gering. Darüber hinaus können projektübergreifende Coding Issues nicht-funktionale Anforderungen haben, um die Quality-of-Service Eigenschaften zu verbessern. Diese Arbeit stellt die Multi-Project Issue Management and Notation vor, eine Modellierungssprache, um Multi-Project Coding Issues und Projekte/Dienste in einem Systemarchitekturgraphen zusammenzufassen. Darüber hinaus wird ein Prototyp eines Multi-Project Coding Issue Management Systems beschrieben. Um diesen Lösungsansatz zu validieren, wird ein Goal-Question-Metric-Plan vorgestellt und eine Expertenbefragung durchgeführt. Abschließend werden mögliche zukünftige Forschungsaufgaben vorgestellt.

Acknowledgement

I want to thank Steffen Becker and Uwe Breitenbücher for their valuable feedback on this thesis as well as general assistance concerning scientific writing. Without your help, the concept of this thesis would certainly have been described in less detail and thought through. Our regular meetings have always been an incentive and motivation to push the work forward. I am looking forward to exploring the outstanding research challenges together with you. I also thank Julian Liedtke and Vanessa Fimpel for proofreading this thesis, especially on such short notice. Finally, I want to thank all industry experts for providing such detailed feedback about my thesis' concept and implementation in order to evaluate it.

Contents

List of Figures

Chapter 1

Introduction

During the last decades, the emerging microservice architecture style gained more and more popularity. As a result, software development turned from a monolithic style towards a service-oriented one. Nowadays, most modern software architectures follow this style in which multiple services build together an application. These flexibly combinable service scan be independently developed and operated by a single team using most appropriate tools and technologies. While development of a new system usually uses an agile process, e.g. Scrum [SS11], the development of a distributed system with multiple teams is more challenging. The most well-established agile software engineering processes are designed for single teams with single projects, and without dependencies to other teams. Hence, developing a complex system consisting of multiple teams and services/projects comes with some difficulties since tasks or issues must be communicated and synchronized over several teams. Sometimes, one team does not know about the other. Communication within one team seems feasible. There are some approaches for multiple teams, like Scrum of Scrums or Nexus, which are suited for a distributed system project, like a microservice application. However, they usually have some communication overhead, e.g. through additional meetings, or e-mails.

1.1 Problem Statement

The overall problem statement is to synchronize issues, tasks, and artefact, e.g. interface definition models, between multiple projects and teams. There are two parts of the problem statement that are addressed in this thesis. The first problem statement is communicating issues concerning multiple projects or teams in a qualitative way using coding issues. This is stated out in Section 1.1.1. Afterwards, Section 1.1.2 describes the problem statement of synchronizing model and design decision changes between multiple projects or teams.

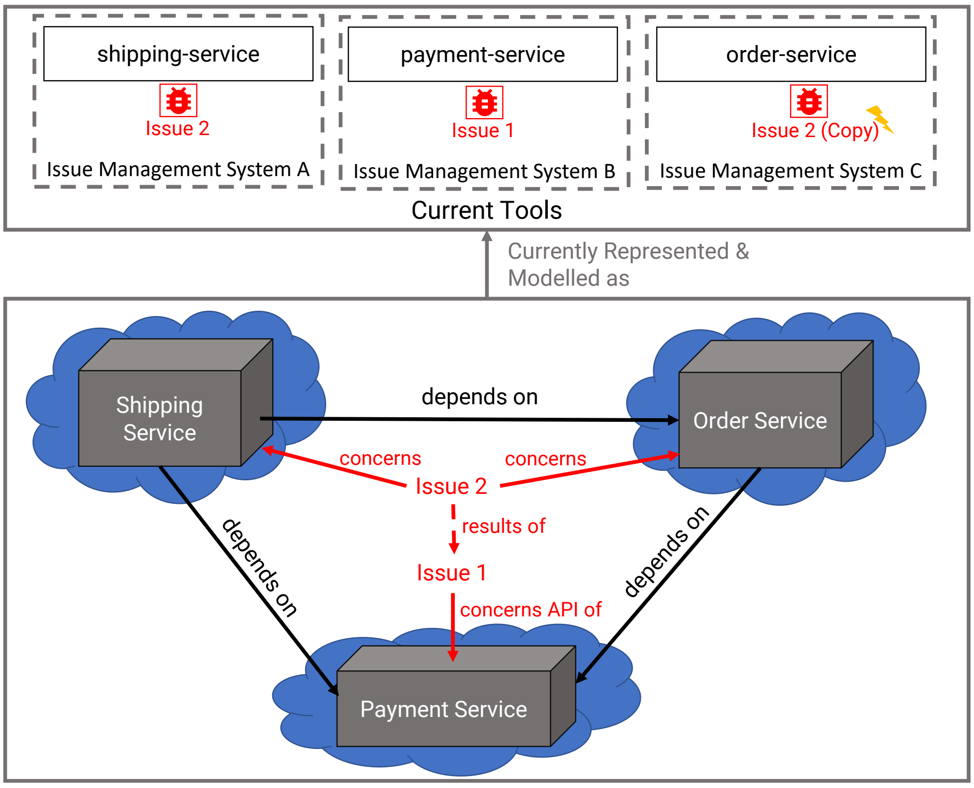


Figure 1.1: Current approach to communicate issues concerning multiple projects using coding issues

1.1.1 Dealing with Multi-Project and Multi-Team Communication of Issues using Coding Issues

Usually issues affecting multiple teams and multiple projects are communicated with e-mail, instant messengers, (video) calls, comments for coding issues, and additional meetings. However, this is not very effective since it has some communication overhead as mentioned above. Tasks or issues concerning a single project (service) often are communicated using coding issues. While this works well within one project this approach is not feasible for multiple projects since coding issues are usually stored locally for each project. Therefore, developing a complex system with more than one project results in coding issues stored in several locations which requires synchronization of those coding issues.

Existing state-of-the-art solution attempts mentioned in several forum posts in Jira or Redmine forums[1][2][3] for synchronizing these coding issues between the projects are more work-around solutions than suitable. There are a lot of solution attempts mentioned. Often, the coding issue is copied to all other relevant projects as shown in Figure 1.3. Any change on the coding issue must be propagated through all other occurrences too. These coding issue clones are prone to errors if any duplicate is forgotten to be updated. Therefore, technical debt arises. Another approach is to create a new coding issue containing the issue's problem statement in a separate, shared location, and one coding issue for each project containing a URL link to the shared one. As a result, no further synchronization must be performed. However, a developer always must follow at least one link to gather all information about the issue. Therefore, a system to manage coding issues concerning multiple projects and teams is needed, so neither coding issues must be duplicated, nor links must be followed.

1.1.2 Synchronizing and Communicating Artefact Changes concerning Multiple Projects

Services in a microservice architecture usually access other services' interfaces via messaging, REST, or other technologies. These dependencies are stable if the interfaces do not change. However, during development of a service the interface might change sometimes which concerns dependent services. Therefore, changes in artefacts such as a service's interface or other models which concern multiple projects must be communicated and synchronized between all affected teams and projects as shown in Figure 1.3. Current solution approaches use semantic wikis or (semantic) traceability links towards artefacts and design decisions in order to synchronize artefacts, and communicate their changes within a project. These traceability links can be produced in various ways, e.g. through semantic and syntactic analysis of source code, models, and descriptions. Often, an ontology of the domain is created and used for these links too. However, when it comes to a distributed system consisting of multiple independent projects there is no established approach to there is no established approach to store and manage such links. Hence, communication of the artefact changes does not really occur. Therefore, a system to create, store and manage traceability links towards artefacts and design decisions is needed.

[1]https://community.atlassian.com/t5/Jira-questions/How-do-others-work-with-issues-affecting-multiple-projects/qaq-p/399950
[2]https://community.atlassian.com/t5/Jira-Software-questions/Share-one-issue-quot-ticket-quot-across-multiple-projects-and/qaq-p/407534
[3]http://www.redmine.org/boards/1/topics/21939

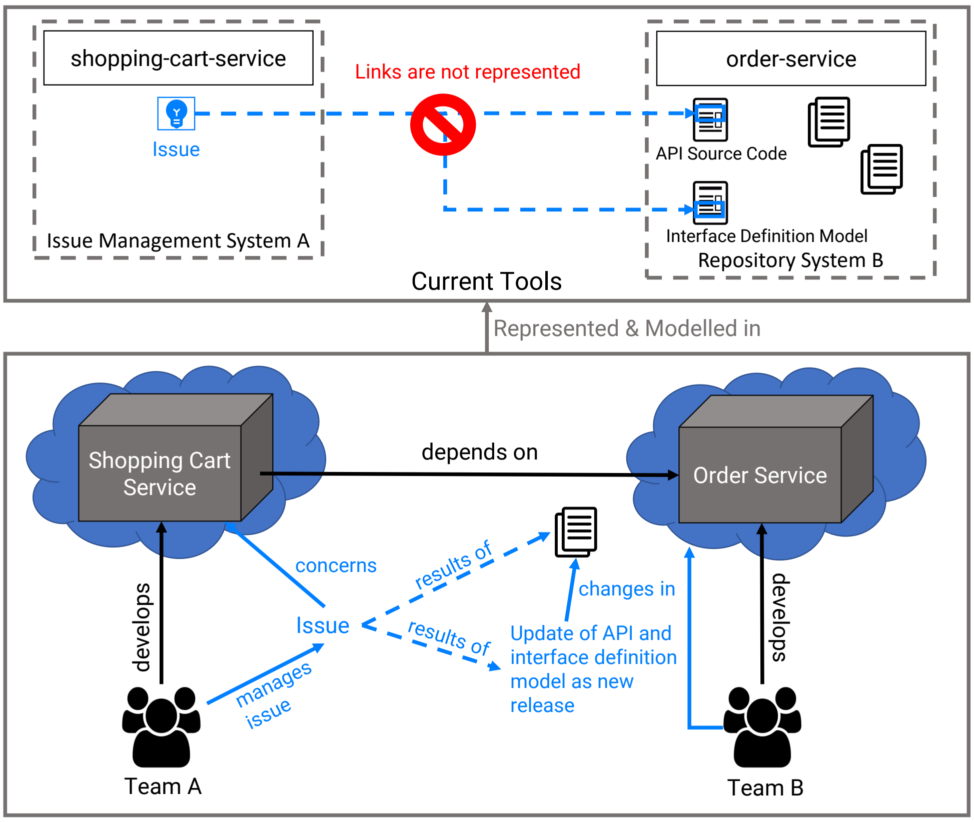


Figure 1.2: Current approach to communicate artefact changes using coding issues

1.2 Solution Approach

This thesis presents the concept of multi-project coding issues. Multi-project coding issues can, in contrast to normal coding issues, concern multiple projects and, therefore, multiple teams at the same time instead of one as shown in Figure 1.3. As a result, none of the work-around solutions mentioned above is needed. Each developer of at least one of the concerned projects has to see the original coding issue. These multi-project coding issues are extended optional links to other (multi-project) coding issues. Instead of providing only a URL in the coding issue's body text, linked coding issues can be shown directly in the multi-project coding issue. As a result, multi-project coding issues can represent dependencies between issues that would normally not be directly depictable without additional data. A multi-project coding issue can have multiple meta-data. In order to synchronize and communicate artefact changes, e.g. a service's interface source code, between several teams fine-grained semantic traceability links, which were mentioned above, can be added to multi-project coding issues. Additionally, a developer can include non-functional requirements to a multi-project coding issue. The idea behind these non-functional requirements comes from WS-Policy and, therefore,

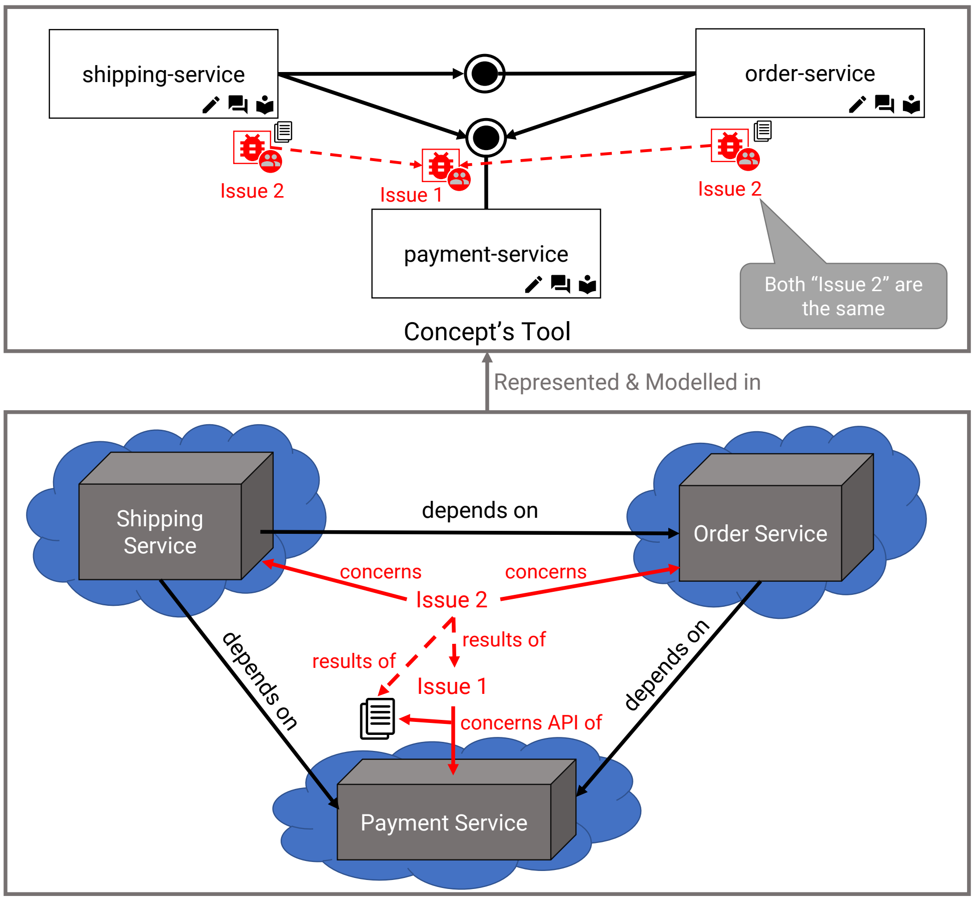


Figure 1.3: This thesis's solution approach for the problem statement

they build contracts that must be fulfilled in order to solve a multi-project coding issue. For example, in case of a service interface change, an average response time can be defined as constraint. Monitoring systems can be used to check whether a non-functional requirement is fulfilled.

Since multi-project coding issues might concern several projects and have dependencies to other coding issues, a simple list of coding issues for each project is not suitable anymore. A better way to visualize multi-project coding issues for a complex microservice application consisting of several projects could be as a system architecture graph where the projects/services and their interfaces are nodes, and their dependencies edges. Multi-project coding issues can be pinned as nodes to the project/service nodes. As a result, affiliations of multi-project coding issues, and their dependencies can be visualized in a qualitative way. For this notation, this thesis introduces a modelling language that maps such a system architecture graph. Furthermore, a multi-project coding issue management system is described which implements multi-project coding issues, and this modelling language.

In order to evaluate and validate this thesis' concept, A Goal-Question-Metric plan is created, and an expert survey performed to show that the problem has been addressed and might be a good solution approach for it. On basis of this evaluation this thesis' concept is validated. However, there are some threats to validity which are discussed.

To summarize, the contribution of this thesis is the introduction of multi-project coding issues, and a modelling language to notate them together with affected projects/services of a microservice architecture.

1.3 Thesis Structure

The thesis is structured as follows:

Chapter 2 – Foundations and Related Work surveys the foundations and related work this thesis is based on.

Chapter 3 – Concept of Multi-Project Coding Issues: This thesis' concept's requirements, aspects of the concept, domain model for multi-project issue management, and the concept of a graphical notation for issue management are introduced.

Chapter 4 – Implementation: In this chapter, the implementation, used tools and technologies, and especially the system architecture graph editor are shown.

Chapter 5 – Validation describes the validation of this thesis. First, the used measure for validation is described. Afterwards, this chapter explains the process of how the feedback was gathered, summarizes the results of the feedback, and analyses the outcome. Finally, threats to validity for this thesis are outlined.

Chapter 6 – Conclusion and Future Work concludes the results of this thesis and shows future research challenges.

Chapter 2

Foundations and Related Work

This chapter explains foundations and related work for this thesis. First, the foundations are outlined in Section 2.1. Afterwards, Section 2.2 presents related work regarding the problem statement and the systematic survey procedure how related work was gathered. The related work itself is divided into two parts. The first part describes related work regarding linking models and notation. Finally, current approaches for synchronizing cross-team communication, and issues are presented.

2.1 Foundations

This section describes foundations used in this work. First, Section 2.1.1 explains coding issues as they are the core concept part of the thesis' implementation. Afterwards, issue management systems are described in Section 2.1.2, and repository system are delineated in Section 2.1.3. Finally, Section 2.1.4 outlines briefly what a microservice architecture and a service-oriented architecture are, since this work concentrates on those architecture styles as particular use case.

2.1.1 Coding Issues

As central part of change management in software engineering, coding issues are a state-of-the-art way to protocol feature requests, change requests, bug reports, defects, or other kinds of issues related to the development of a software[1]. Therefore, they provide crucial information about the source code they are relating to or features to be

[1]https://help.github.com/en/github/managing-your-work-on-github/about-issues

implemented to developers [BJS+08]. Usually a coding issue contains a title, which describes shortly the content of the issue, a body text containing a detailed description of the issue, a creator (reporter of the issue), and some meta-data, e.g. developers which are assigned to solve the issue or labels to classify the issue [Som11, p. 744-745][LL13, p. 576-578]. Coding issues are managed by the developers, product owner or other stakeholders of a project. They are stored in an issue management system (cf. Section 2.1.2). In some issue management systems additional meta-data can be added to the coding issue. Coding issues always have a state which tells whether they are open or closed. Closed coding issues are either already solved or not relevant anymore.

The body of a coding issue is a plain text field in which a user of the issue management system can write any text. Formatting like markdown often is enabled[2] for a better readability. There are good practices how to write down coding issues so that they are easy understandable[3][4][5]. Usually a coding issue is related to a part of source code or model, and often to a commit.

2.1.2 Issue Management Systems

Within the scope of this thesis an issue management system is defined as a system which is designed to handle

issues as coding issues. An issue management system, therefore, allows users to create, store, manage, edit, etc. coding issues. Furthermore, an issue management system can help developers creating coding issues providing templates, reminders, and contextual awareness [JPZ08]. Issue management systems are often called bug tracking system too. Since not only bugs but also other types of coding issues are managed within the scope of this thesis, the term issue management system is more appropriate. Usually, but not necessarily, an issue management is connected to a repository system (cf. Section 2.1.3), in which the program code of the software project is stored.

Common issue management system providers are GitHub[6], GitLab[7], Redmine[8] or Jira[9], however, there are other providers too. In some cases, e.g. GitHub or Gitlab, there is only one system working as issue management system and repository system at the same

[2]https://www.markdownguide.org/basic-syntax/
[3]https://developer.mozilla.org/en-US/docs/Mozilla/QA/Bug_writing_guidelines
[4]https://qablog.practitest.com/principles-of-good-bug-reporting/
[5]https://www.ministryoftesting.com/dojo/lessons/the-art-of-the-bug-report
[6]https://github.com/
[7]https://about.gitlab.com/
[8]https://www.redmine.org/
[9]https://www.atlassian.com/de/software/jira

time. The features of each issue management system differ from provider to provider. However, most providers offer a possibility to add coding issues to milestones.

Another well-established possibility to organize coding issues is a project board. Providers like GitHub offer projects[10] where developers can manage coding issues in project boards like Kanban boards [HS14]. Those boards can show issues in different columns. Each column has a specific meaning, e.g. it contains all coding issues of the current milestone which are in progress. The columns are usually configurable, and can have some automation for the coding issues.

Additionally, some issue management systems, e.g. Gitlab, provide evaluation and analysis of the coding issues. Those evaluations can be for example burndown charts[11], a visual measurement tool to describe the progress of a project graphically. A burndown chart shows the remaining effort, e.g. number of remaining open issues, in relation to the remaining time of the overall project or current milestone. So e.g. the amount of work project developers deliver in each milestone can be computed through such evaluations.

Time tracking can be crucial for every software project. To know how much time for a coding issue is planned is a good start. However, due to several reasons sometimes coding issues need much more time than expected. This usually leads to less time for the work on the remaining coding issues. In order to get a good overview of the time management of a current milestone or iteration, issue management systems like Gitlab[12], or Redmine[13] provide time tracking tools either already build in or as plugins. Developers can look of the needed and remaining time. Moreover, they can analyse which kind of coding issues were not accurate enough in their time forecast.

Notifications help developers to recognize any essential event. Therefore, all issue management systems mentioned above provide e-mail notifications for several events, e.g. a coding issue is closed or commented. Those notifications can be enabled or disabled for every developer on his own.

While issue management systems provide, like GitHub, only a small amount of access rights settings for developer, other issue management systems, like Redmine, enable to manage rights on a fine-grained basis for each developer individual. Possible right settings can be (1) full-fledged write access, where developers can write, edit, comment, and view coding issues, (2) create, view, and comment rights, but no editing, (3) only

[10]https://help.github.com/en/github/managing-your-work-on-github/about-project-boards
[11]https://docs.gitlab.com/ee/user/project/milestones/burndown_charts.html
[12]https://docs.gitlab.com/ee/user/project/time_tracking.html
[13]https://www.redmine.org/projects/redmine/wiki/RedmineTimeTracking

comment, and view or (4) only view rights. Other combinations can be supported too. However, there are issue management systems where only a few settings are possible.

2.1.3 Repository Systems

Within the scope of this thesis a repository system is defined as a system which are using software for distributed version control like git [LM12; Spi12] or subversion (SVN) [PCSF08] to manages source code repositories. Those source code repositories contain source code of software projects. Common git providers are GitHub and GitLab. In most cases, the providers offer not only single code repositories. Rather, for example in GitHub, organizations can be created which are a collection of several code repositories[14]. Developers can be members of organizations and, therefore, get access to the organization's code repositories. A GitLab or GitHub code repository can show which developers contributed to it and gives analysis of how many lines of codes and commits each one has written. Additionally, for example GitHub organizations can divide the developer members into teams where each team can get access to their own code repositories granted[15].

In git repositories developers can create branches of a source code version, develop in their branch, and after completing their task, they can merge their own branch with the joint default branch (usually called master branch). Often, before merging the updated code with the default branch, a developer must create a pull request. This pull request describes the changes the developer made in respect to the default branch, contains the updated code files, and should be reviewed from other developers. The developers then can decide whether to accept the pull request and merge the branches, or deny it. Providers who are combining issue management systems with repository systems are providing the possibility to add coding issues to pull requests. This allows to link coding issues to specific commits. A developer can look which coding issues have been resolved with a certain pull request.

Distributed version control systems offer a commit history where a developer can see which change was fulfilled at which time step. Moreover, such a history allows a developer to go back to a previous version of the source code to check these files out. This can be very helpful in case some code was broken and could not be repaired. Then, a developer can return to the last point where it worked, and starts over from there. If

[14]https://help.github.com/en/github/setting-up-and-managing-organizations-and-teams/about-organizations

[15]https://help.github.com/en/github/setting-up-and-managing-organizations-and-teams/organizing-members-into-teams

the commit history is linked to coding issues the developer can identify which issues led to the problem, and reopen these coding issues again.

2.1.4 Microservices and Service-Oriented Architecture

This section is based on a German version of my bachelor's thesis which is publicly available on the web page of university library of the University of Stuttgart [Spe17]. The original text has been summarized here to approximately one third of the original size. Some sentences could be direct translations of the German version.

The emerging and hyped microservice architecture style [Fow17; New15] gained more and more popularity during the last years, and is discussed a lot. However, no clear definition exists, what a microservice is. The software engineer and one of the original signatories of the agile manifesto[16] [FH+01], Martin Fowler and his colleague James Lewis describe the microservice architecture style as an approach for development of a single application in the form of a series of small services, which are running in their own process and communicating through easy mechanisms, e.g. HTTP-resource-based APIs [FL15]. Those services are based on business functionality and are independently updateable and exchangeable through fully automated deployment. Each microservice in a microservice architecture can be deployed on its own by an independent developer team. In contrast to a monolith architecture, microservices are no developed as a single big block. Until 2016, there was not a generally accepted or established opinion of how big a microservice should be [FLW16]. According to Fowler, the microservice architecture style follows the principle of "smart endpoints & dump pipes" [Fow17], which means that the middleware for communicating between and with microservices is kept as simple as possible. As a result, more complexity must be shifted to the API endpoints.

A microservice architecture is a special form of a service-oriented architecture, which is an architecture style that assembles systems from networks of distributed services. Those service can be used to implement business functionality [LL09]. The idea is to develop many reusable services in order to enable the most flexible possible composition of new applications. The smaller services there are, the more likely it is to be able to reuse parts of the services without having to re-implement the logic. A service in the service-oriented architecture must provide standardized interfaces with which other applications can interact. The difference between service-oriented architecture and a microservice architecture seems not really to exists. Martin Fowler, on the other

[16]https://agilemanifesto.org/

hand, describes the architectural style of microservices as a subset of the use of service-oriented architecture. The microservice architecture style needs that every service should be deployable and usable on its own. This requirement is not necessarily given in service-oriented architecture.

2.2 Related Work

In the following, the current state of the art with respect to cross team communication and synchronization of issue tasks is reviewed. First, this chapter outlines research papers working on ways of creating links between issues, design decisions, and other artefacts in order to improve deployment of an application and reduce communication overhead. Afterwards, it describes how the industry deals with the problem of synchronizing issues between several teams and states out the problems their approaches have.

2.2.1 Survey Procedure

The related work described in this chapter have been surveyed using a systematic literature review process in order summarise benefits and limitations of a specific approach in an unbiased and repeatable way. In addition to this, gaps in current research can be identified. As a result, possible areas for further investigation can be determined and positioned in respect to existing researches. For performing this systematic literature review the well-established Kitchenham's method [KBB+09; KPP+02] is used. These guidelines contain three phases. The first phase is planning the review in which survey questions are specified As second phase the review is conducted which means research is identified. From this set of identified research primary research is selected and its data extracted. The gathered papers were narrowed down in three steps. First, research papers were excluded by their title or abstract. Afterwards, introduction and conclusion were further exclusion criteria. Finally, figures, tables, and section overviews were scanned before classify the paper as relevant and analyse it in depth. The last phase is reporting the review after an evaluation of its data.

It is crucial in any systematic literature review to ask the right survey questions which means they should be meaningful and important to researchers and industry. For this thesis, the following survey questions were specified:

(SQ1) How can issues affecting multiple projects or teams be communicated?

(SQ2) How can model changes be synchronized over several projects or teams?

To collect the related research papers, the search engine Google Scholar[17] was primarily used since it contains publications and articles from most important digital libraries in computer science. Additionally, snowballing was performed on reference lists of relevant research papers. These reference lists have been inspected for more relevant papers.

The following key words were used in search engines either on their own or in combination where appropriate: multi-project issues, multiple projects shared issues, synchronizing shared issue multiple teams, multiple software projects, linking design decisions, issues linking, issues affecting multiple projects, issues affecting multiple teams, relating issue to multiple projects, synchronizing model changes, synchronizing design decisions, issues traceability links, synchronizing cross-team communication, communicate service interface changes.

2.2.2 Linking Models and Documentation

The increasing communication overhead and technical debt through use of deprecated models is a big issue in software engineering. Zhang et al. try to solve it with an automatic establishment of traceability links between code and documentation. They are using NLP for text mining to retrieve structural and semantic information, that can be found in the artefacts. Using this information, the traceability links can be created. Rather than using complete documents and code files, Zhang et al. are trying to build those links between individual words and code entities. However, no further semantic analysis is executed to increase e.g. correctness of those links. Instead they analyse the abstract syntax tree of the code and identify entities and relations. Using information, given by the abstract syntax tree and the structural and semantic analysis of the documentation, a linked ontology is built. This ontology encodes knowledge of the software domain in a formal language. As an implementation, Zhang et al. propose an Eclipse based system for their approach. Nevertheless, their paper is very high level and does not contain much details how they are creating the ontology [ZWRH08]. While Zhang et al. trying to link code and documentation of one project to reduce communication overhead and error-prone of models becoming a different state than the code, they have not considered multiple teams or multiple projects connected to each other.

Maalej et al. pay attention of distributed systems. They acknowledge an overhead for documentation of changes and communication in distributed systems and want to address this issue. That is why they state out that traditional knowledge management fails, when it comes to multiple teams and multiple projects. Since normal wikis have

[17]https://scholar.google.com/

prevailed successfully in conventional software development, their approach is to use wiki structure as well for documenting and communicating changes and other issues. However, information in normal wikis is often unstructured. Hence, they propose a combination of semantic web and normal wikis to get a more structured kind of wiki. They call this kind of wiki a semantic wiki. Like Zhang et al., they are creating an ontology of documents and code with the semantic wiki. In addition to this, they annotate the wiki by providing meta-data for existing features through additional resources. In contrast to Zhang et al., Maalej and his team are trying to create the ontology with a more context-based than text-based content. So, Maalej et al. use the semantic wiki as coding issue and bug tracker [MPH08]. Yet, it is inconvenient for modern software developers to track their feature requests and bug report coding issues in a wiki. Usually coding issue management systems like Jira, Redmine or GitHub are used.

Another approach is to link design decisions to design models to better synchronize changes. Könemann and Zimmermann propose such an approach, in which they bind the outcome of a design decision to models. Then they store the new model and the model differences as well. To get the design decision, they have created a meta-model of a typical design decision, depicted in Figure 2.1. In their meta-model, they describe a design decision consisting of an issue, some alternative solutions, and some outcomes for the given issue. Each issue can have various alternative solutions which are related to the issue, but at most one of them is chosen as outcome solution. An outcome has a status, assumptions, some justification, why the solution was chosen and consequences of the solution. There can be zero or more outcomes of one issue. Könemann and Zimmermann recognize that such a meta-model is implemented in some design decision management tools, but none of them includes the design model changes. Due to this, they want to store not only design models, rather they store the model changes and different versions of the models to better document the design decisions. In addition to this, they use the models and design decisions for a consistency check at design time. For that purpose, they create constraints which have to be fulfilled. The user of their system gets notified in case some constraint is unsatisfied. Finally, Könemann and Zimmermann are reusing the design decisions partially automated by adding design model changes to design issue solutions. Their concepts are tool-independent to be most reusable [KZ10].

Capilla et al. are linking architectural design decisions towards artefacts in the software engineering life cycle. They state that design decisions must be linked fine-grained to control modifications in design. Furthermore, they outline an importance of recording design decisions in case any expert of the team would change, so new team members will understand the design decisions. This should reduce maintain-effort. During their work, they observed several prototype tools to support design decisions and recognized, that the link granularity often seems to be too course-grained for an appropriate use. Capilla

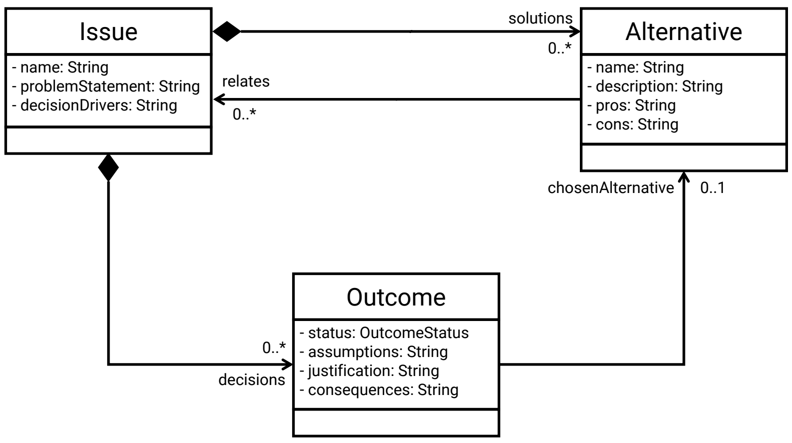


Figure 2.1: Metamodel of a design decision [KZ10]

et al. suggest versioning of models and logging of decisions to gain better comprehension of the decisions. Moreover, they outline that quality-of-service conditions and other run-time decisions should be supported as well in design decisions. As main contribution, Capilla et al. are explaining their meta-model for architectural design decisions, which is implemented in the Architectural Design Knowledge Wiki tool. Finally, they add an analysis of a user survey they conducted to validate their meta-model [CZZ+11].

2.2.3 Synchronizing Cross-Team Communication and Issues

It seems, there are some improvement opportunities regarding research tools. However, the industry must deal with the problem of cross team communication and synchronization as well. Using an expert survey, solution attempts how industry experts are communicating issues to other teams and synchronize their work could be gathered. The industry experts indicated various solution attempts. Current approaches are making use of additional meetings, more e-mails, instant messaging applications, e.g. Slack or Microsoft Teams, video calls or GitHub coding issue comments. However, sometimes the rule in a company is to prohibit any major change idea completely, especially it might be a breaking change. While all these solutions are established in both big and mid-size software companies, they all result in a high communication overhead.

The approach of this thesis is to extend a commonly used tool, such as coding issues, to enable cross-team and cross-project coding issues, which can be supported with traceability links and non-functional requirements for an improvement of quality-of-service and reduction of communication overhead between several teams. During the last few years, there seems to be some interest in several forums of Jira[18] or Redmine[19] for such a tool. In the first question in the Jira forum, a developer asks how others work with issues affecting multiple projects. For better understandability of the problem, the developer adds a description of his problem. As a first solution, he mentions himself that he could clone the coding issue. However, he states out, that this approach "seems like a big overhead in maintaining their (the coding issues') state and not a good system since the clones do not get updated information"[20]. If any change occurs, this change must be propagated through every copy of the coding issue. The answers discuss possible solutions. Another solution is to create a new coding issue in a shared place and a coding issue for every related project with a link to the shared one. This means that coding issues are not become outdated. However, a developer mentioned that there would be a need to go through links for every coding issue affecting multiple projects. They would like to have some syncing of coding issues between projects. The second question asks if there is some way to share a coding issue across multiple projects and show on both project boards[21]. An additional solution approach for this question works only for if all projects are managed by Jira. They explain that two or more projects can share a board. However, coding issues affecting multiple projects cannot be created there as well. Every of the workaround solution which can be found in the forums seems not to be optimal, so a tool to support coding issues for multiple teams and multiple projects seems to be needed. In addition to the two Jira forum questions, there is a similar question in the Redmine forum where a developer asks if it is "somehow possible to relate an issue to multiple projects"[22].

The software company K15t is specialized on building software for the Atlassian software world to which Jira belongs. One of their products is the Backbone Issue Sync for Jira[23], which enables an automatic synchronization of Jira coding issues between several Jira projects. Instead of syncing a complete coding issue, a developer can specify which data of the coding issue should be synchronized. While this Jira plug-in enables sharing a coding issue between Jira projects, it is not completely suitable for big microservice architectures in which a lot of service projects might not managing their coding issues with Jira.

[18]https://community.atlassian.com/
[19]https://www.redmine.org/projects/redmine/boards
[20]https://community.atlassian.com/t5/Jira-questions/How-do-others-work-with-issues-affecting-multiple-projects/qaq-p/399950
[21]https://community.atlassian.com/t5/Jira-Software-questions/Share-one-issue-quot-ticket-quot-across-multiple-projects-and/qaq-p/407534
[22]http://www.redmine.org/boards/1/topics/21939
[23]https://www.k15t.de/software/backbone-issue-sync-for-jira

Chapter 3

Concept of Multi-Project Coding Issues

This chapter describes this thesis' solution approach, a concept of multi-project coding issues, for the problem statement. First, a requirement engineering process has been performed to gather requirements which is explained in Section 3.1. Afterwards, these gathered functional and non-functional requirements are written down in Section 3.2. Based on the gathered requirements a concept was built to solve the problem statement. Section 3.3 briefly summarizes this concept in an overview. Afterwards, in Section 3.4 the three main contributions of this thesis' concept are presented in detail. Subsequently, Section 3.5 shows the domain model for this thesis' concept. Finally, a modelling language to notate coding issues and software architectures together in a graphical view is introduced in Section 3.6.

3.1 Analysis and Requirements Engineering Process

To analyse the requirements of this thesis' objective, a requirement engineering process has been performed. This requirement engineering process had five activities, which were performed as shown in Figure 3.1. First, some requirements elicitation was done, in which knowledge about the problem domain and requirements were gathered. As part of the requirements elicitation, stakeholders were identified and addressed to gather the information. In addition to this, an internal analysis, e.g. brain storming during meetings, was done. As second activity, the gathered requirements were specified. As part of this activity, a proposal talk, and paper was written for internal approval. The third activity was some requirement validation through expert surveys and supervisor meetings. For analysing, documenting, and agreeing on the requirements, requirement management was performed as fourth activity. As last activity, system mock-ups were created.

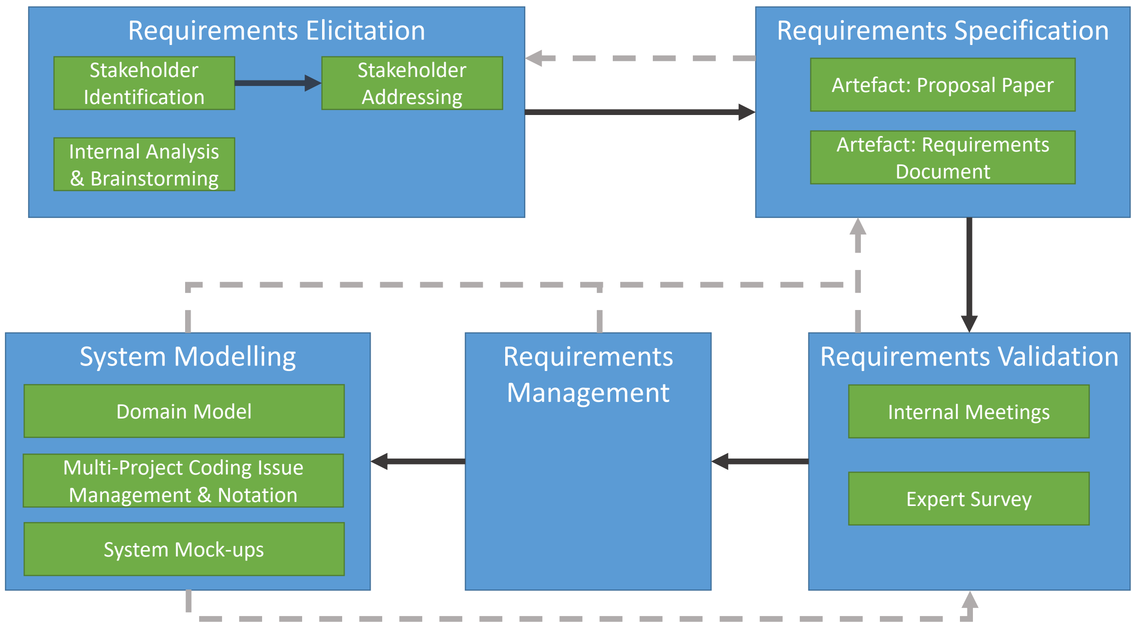


Figure 3.1: Performed requirements engineering process

To start with the requirements elicitation, stakeholders had to be identified. Five types of stakeholders could be identified for this thesis: (1) domain experts who give essential background information about the system application domain to solve the problem best, (2 - 4) software developers, software architects and project managers/owners as end user for the thesis' system and (5) the thesis' supervisors to preserve the scientific context of this work. It must be noted that a stakeholder can fit into several of these types. For each of the first four types of stakeholders some colleagues which are working in this field were asked. They were contacted over E-Mail or instant messengers like Telegram. To address the stakeholders best, use cases containing the problem statement were created and provided. Some stakeholders were also contacted and addressed in personal. An internal analysis of the requirements was carried out with the thesis supervisors at regular intervals and adjusted or updated if necessary. In brainstorming sessions during the meetings some features and changes of the requirements were made.

As part of the requirements specification activity, the first requirements were specified in a proposal paper which was presented during an intern proposal talk. Researchers from the examiner's institute as well as invited guests were present to critically review the requirements and the solution concept. The resulting feedback was then included in a second version of the proposal paper. As the work progressed, the changes in requirements, which discussed in the meetings, were recorded on paper. Between and during the meetings a formal domain models were built on basis of the requirements.

During requirements specification activity, whenever more knowledge about the problem was required, the elicitation process was triggered again.

The requirement validation activity for the thesis' solution concept was performed in two steps. In order to meet the requirements of the supervisors, regular meetings were held to discuss and review the current state of development of the concept and implementation regarding the requirements of the supervisors and experts. In addition, an expert survey was conducted to evaluate the concept of the work for some validation. Some evaluation was performed earlier to ensure that errors in the requirements definitions would not propagate to the successive stages resulting in a lot of modification and rework.

Frequent meetings with the supervisors were an important part of the requirements management activity. During these meetings updating and prioritisation of the requirements as well as agreeing on them was done. Using these brainstorming sessions we modified the thesis' concept in a systematic and controlled manner whenever it seems necessary.

The system modelling was performed in parallel to the other activities. As part of this activity, several phases were completed. In each phase a part of the concept was modelled and specified. First, the domain model was created in several steps. Each version was reviewed by the thesis' supervisors. After having a suitable domain model and some first experience of a prototype, the multi-project issue modelling and notation was modelled. Finally, the system's mock-ups were created which then were used for an expert survey to validate the concept and implementation.

3.2 Gathered Requirements

As output of the requirements engineering process, several requirements were gathered. This section points out the gathered requirements, mostly written as user stories. It must be noted that a *user* in this user stories can be every kind of user for this thesis' multi-project management system, e.g. software developer or project owner/manager. First, all functional requirements regarding the multi-project issue management and notation are described.

- As a user, I want to create a coding issue concerning one or more projects/services (multi-project coding issue).
- As a project manager/owner, I want coding issues finer grained classified to feature requests and bug reports to better manage the software development status of a milestone and assign critical issues faster.

- As a user, I want to add non-functional requirements/constraints to a multi-project coding issue to enable e.g. monitoring for automatic check of the requirements and reject of pull requests. A non-functional constraint could be for instance the maximum run-time of a request against an interface.
- As a user, I want to link multi-project coding issues to other multi-project coding issues (which could be for other projects/services) to create traceability links between coding issues and a dependency view of them.
- As a user, I want to link multi-project coding issues to additional artefacts to better clarify the scope and goal of the issue. Such additional artefacts can be for instance parts of programming code files or interface definition models.
- As a user, I want to divide coding issues for projects/services and for service interfaces, to identify faster the concrete location of issue.
- As a project owner/manager, I want to have a summarized view of all projects/services and coding issues in a graph notation, to get a fast and qualitative overview of the entire system.
- As a software developer, I want to have view of all projects/services and coding issues to see relations and dependencies between projects and coding issues.
- As a user, I want coding issues which are concerning multiple projects or services be managed easily in one view to fast address all stakeholders of the issue and avoid technical debt of coding issue clones.
- As a user, I want to see the links between coding issues in a graphical view of the project/component composition.
- As a user, I want to have two views. One graph view containing projects/services, service interfaces and coding issues and a list view for a detailed list of all projects/components with their coding issues.
- As a multi-project project owner, I want to add and remove projects/services and service interfaces.
- As a multi-project project owner, I want to compose projects/services through service interfaces or directly.
- As a user, I want to filter for feature requests, bug reports or unclassified coding issues in the graph view and list view.
- As a user, I want to filter for notifications of coding issues in the graph view and list view.

- As a user, I want the system to support several provider types of issue management systems for storing and managing coding issues.
- As a user, I want to have no need for UI switches. All operations for a coding issue should be able through the system and propagated to the corresponding issue management system.
- As a user, I want to store a multi-project coding issue in every corresponding project's/service's issue management system and updated through the thesis' system.
- As a user, I want to close multi-project coding issues if I have the necessary rights for it.
- As a user, I want to edit a multi-project coding issue. If I do not have the necessary rights to edit the original coding issue, I want a shadow to be created.
- As a user, I want to comment a multi-project coding issue, if I have the necessary rights for it.
- As a user, I want only to see the shadow of a coding issue, if it exists.
- As a user, I want my changes in a shadow be propagated to the original coding issue, if possible.
- As a user, I want to add labels to a multi-project coding issue.
- As a user, I want to add software developers as assignees to a multi-project coding issue.

In addition to the functional requirements above, some functional requirements for the overall system are gathered too.

- As a user, I want to create an account, giving a username and password.
- As an organization, I want to have an organization account.
- As a user, I want to be able to belong to zero, one or more organizations.
- As the system, I need the users to provide account credentials for every issue management system they use, e.g. GitHub, Gitlab, Redmine or Jira.
- As a user, I want to sign-in and sign-out.
- As a project owner/manager, I want to create a multi-project project.

- As a project owner/manager, I want to create a multi-project project in the name of my organization.
- As a multi-project project, a project repository and issue management system need to be provided.
- As a multi-project project owner, I want to add collaborators to a multi-project project.
- As a multi-project project owner, I want to remove collaborators from a multi-project project.
- As a multi-project project collaborator, I want to be added as collaborators to the project's repository too.
- As a user, I want to be able to see my rights for every project/service. Possible rights can be writing access, comment rights, read rights in different combinations.
- As a user, I want to access the system as web application through any web browser.

There are also some non-functional requirements for product quality based on ISO25010's quality model [Iec11] which are described in the following:

Functional Suitability The system should cover most of the specified tasks (functional completeness). The functionality should provide correct results (functional correctness) and facilitate the accomplishment of specified tasks and objectives a lot (functional appropriateness).

Performance Efficiency The system should have appropriate response and processing times, when performing its functions (time-behaviour).

Compatibility There should be a scalable back-end and a front-end so multiple users can perform the systems functions efficiently in parallel while sharing a common environment (co-existence). Multiple clients should be able to interoperate together (interoperability).

Usability The system should be easy to operate, control and appropriate to use (operability).

Reliability The system should be operational and accessible when required for use (availability) and recoverable in case of some errors (recoverability).

Security The system should prevent unauthorized access to, or modification of coding issues and other system's data (integrity). In addition to this, accounts should be authorized through web tokens to identify a user and prove to be the one claimed (authenticity).

Maintainability The system should be built modular (modularity) and modifiable (modifiability) to change components with minimal impact to other components and without introducing defects or degrading existing product quality.

Portability The system should effectively and efficiently be adapted for different or evolving hardware (adaptability), easy to install and uninstall (installability) and should be able to replace another specified software product for the same purpose in the same environment (replaceability).

In addition to the non-functional requirements for product quality, there are also some non-functional requirements regarding quality in use based on ISO25010's quality model [Iec11] for the system to be developed which are described in the following:

Effectiveness A user should be able to achieve specified goals complete and accurate.

Satisfaction The system should be useful. A user should have confidence that the system will behave as intended.

Regarding these requirements, several scenarios were created. Considering two services, e.g. a shopping cart service and an order service. A multi-project project should develop the shopping cart service which depends on the order service. There are two scenarios of particular interest: (1) a developer of the multi-project project has write access for both services and (2) a developer of the multi-project project has only write access for the shopping cart service. The system should be able to handle the following use cases.

- The multi-project project developer creates a bug report on shopping cart service
- The multi-project project developer creates a bug report on order service
- The multi-project project developer creates a bug report on edge between shopping cart service and order service
- The multi-project project developer creates a bug report on shopping cart service and relates to a coding issue on order service
- The multi-project project developer creates a feature request for shopping cart service

- The multi-project project developer creates a feature request for order service
- The multi-project project developer creates a feature request on edge between shopping cart service and order service
- The multi-project project developer creates a feature request on shopping cart service and relates to a coding issue on order service

3.3 Overview of the Concept

The concept deals with a solution idea for multi-project, multi-team issue management in service-oriented architectures. Figure 3.2 shows a use case where two services exists. The first one is a shopping cart service which depends on the second one, an order service. Each service is developed by its own team. During the development process, the order the order service's developer team specifies a new interface. However, the new interface contains a bug in its source code which leads to an error in the running system. As a result, the shopping cart service's functionality breaks. The developer team of the shopping cart service recognizes their own service's functionality breaking and the causing issue. They need to communicate the problem to the developer team of the order service. Using current tools, each service usually uses its own issue management system instance. For example, a coding issue of the shopping cart service could be stored and managed in a GitHub project, while coding issues of the order service could be stored and managed in Jira project. The order service interface is only present in the order service and, therefore, its coding issues stored in order service's Jira project. As a result, coding issues that affect the interface can only be distinguished from coding issues that affect the rest of the service by being described in the title, body, or label of the coding issue. In addition, neither any connection between coding issues of the services nor the dependency of the services on each other can be displayed and recognized. In contrast, the tool of this thesis' concept allows the mapping of the dependency of the shopping cart service to the order service. The interface of the order service is explicitly displayed and can have its own coding issues. Since the system of this thesis is a generalizing wrapper above the actual issue management systems, coding issues are stored by the system in the issue management systems of the respective services. The assignment of a coding issue to an interface can, for example, be achieved by a tag in the coding issue's text body. This tag can be parsed by the thesis' system and then the coding issue can be displayed directly for the service interface.

There are three main aspects for the solution approach. As main contribution the aspect of cross-project coding issues is introduced, which are called multi-project coding issues. These multi-project coding issues build the platform to communicate and manage

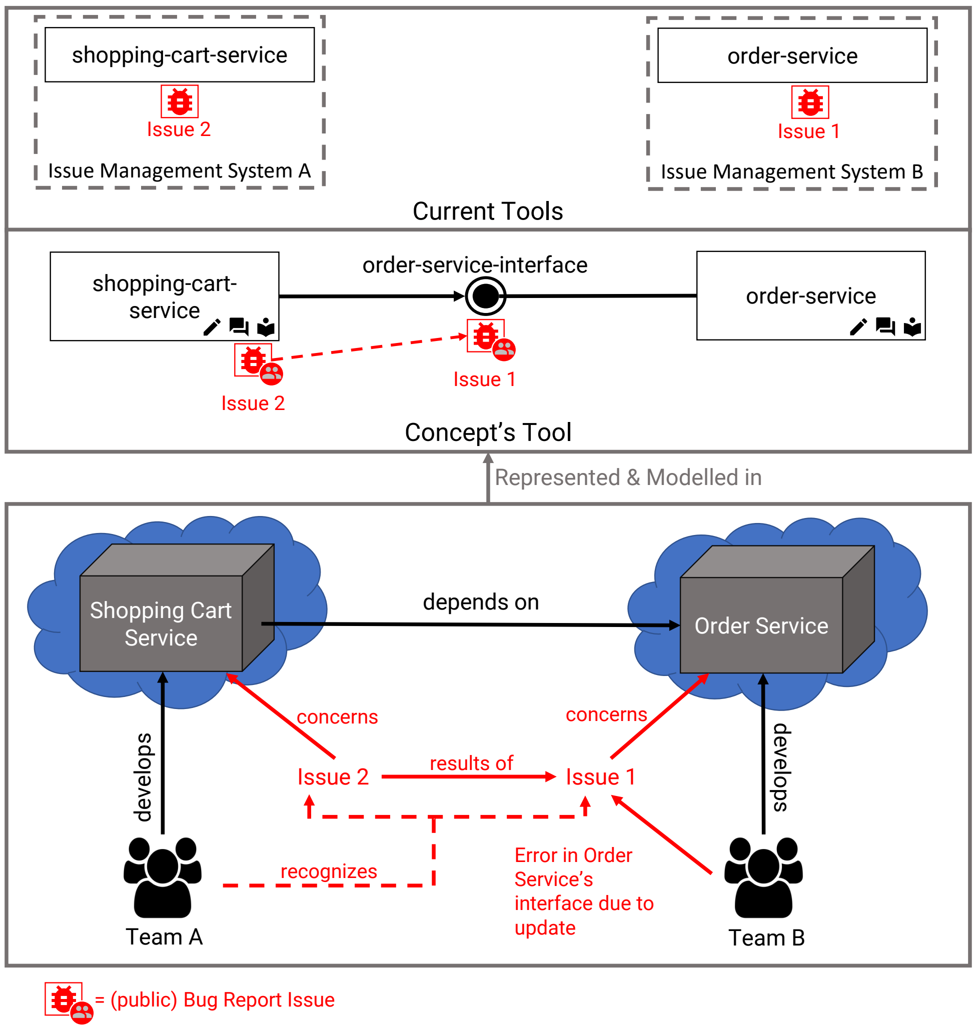


Figure 3.2: The concept's system overview

multi-project and multi-team issues. In contrast to normal coding issues, multi-project coding issues can concern more than one project/service and are stored in all concerned services' issue management systems. Another aspect is the one of traceability links between multi-project coding issues and artefacts. Artefacts can be, for example, lines of code of a source code file or interface definition models. The third main aspect is adding non-functional requirements and constraints to multi-project coding issues. Like WS-Policy this enables automatic checks for coding issues.

In addition to the three main aspects of the concept a domain model for a multi-project coding issue management system is built based on gathered requirements described in Section 3.2. The domain model captures the composition of projects with other projects/services, how the user can interact with the system and how issues are related to concerning issues. Furthermore, the domain model captures where multi-project coding issues are stored.

Another contribution of this work is a modelling language for multi-project coding issues, where a graph notation is used to manage coding issues on corresponding projects/services. This graph notation shows a dependency graph of projects/services to each other by visualizing them using connected edges and service interfaces. The coding issues associated with a project are displayed at the respective project/service node and can link to other coding issues.

3.4 Aspects of the Concept

This section describes the main contribution aspects of this thesis' concept. First, multi-project coding issues are introduced in Section 3.4.1. Afterwards, these multi-project coding issues are extended in Section 3.4.2. This subsection describes how traceability links to artefacts works. Finally, Section 3.4.3 delineates non-functional requirements as quality-of-service contracts for multi-project coding issues.

3.4.1 Multi-Project Coding Issues

The introduction of multi-project coding issues is the main contribution of this thesis. A multi-project coding issue can concern several independent developed and operated projects/services. If the coding issue directly concerns more than one projects or services, it is stored in the issue management system of each project/service. Every change in the multi-project coding issue can be propagated from the system of the thesis' concept to the specific issue management systems and vice versa. Multi-project coding issues can have links to other coding issues which can be located in the same issue management system or other project's/service's issue management systems. The basic usage of multi-project coding issues is to ease cross-team communication of issues in a convenient way through coding issues.

The abstract syntax of the meta-model for multi-project coding issues is shown in Figure 3.3 and described in the following. Like common coding issues, a multi-project coding issue contains a title and body text to detail the problem statement. There are other commonly supported features for a coding issue, such as labels consisting of a name and colour to classify the problem the issue is about and a list of developers as assignees which should solve the issue. A developer is working for at least one project

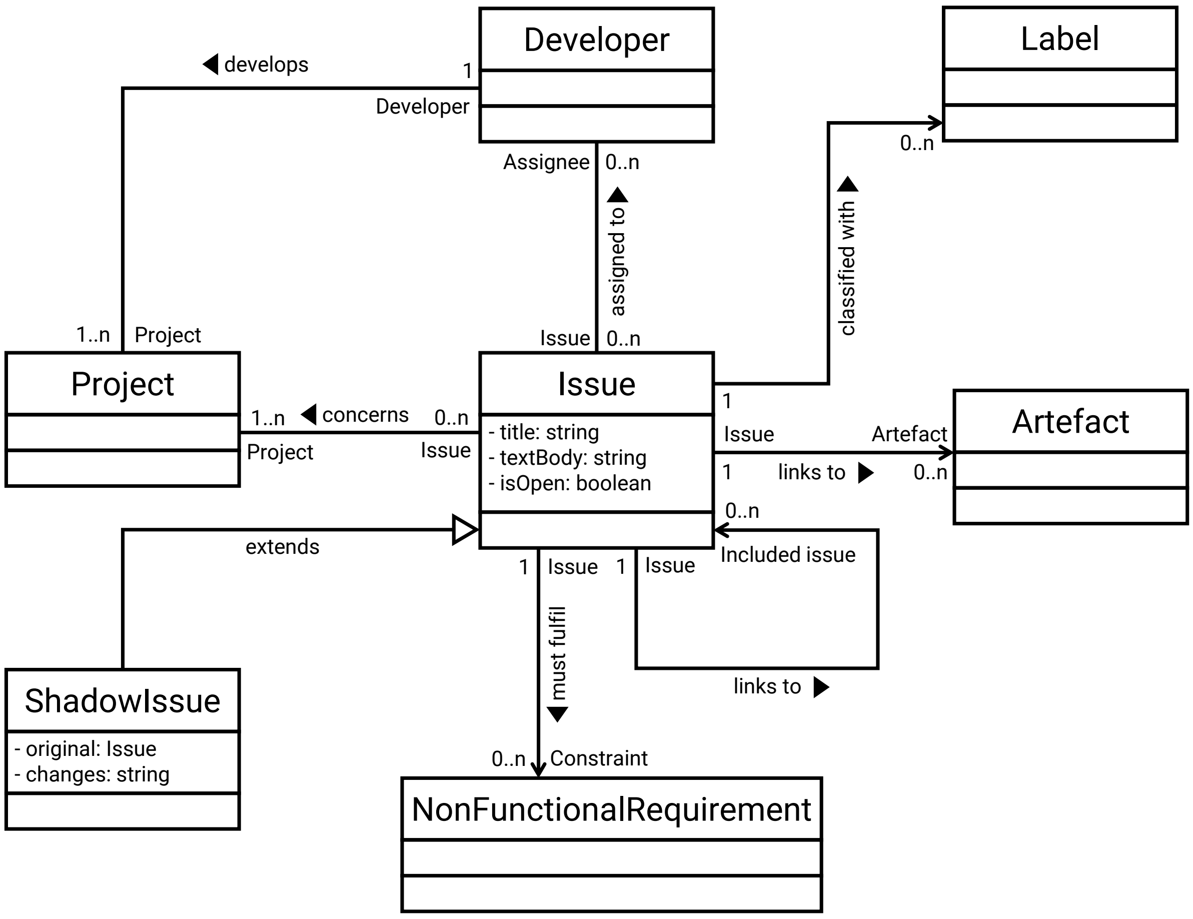


Figure 3.3: Meta-model for multi-project coding issues

which is concerned by the coding issue. In addition to this, a coding issue has a status if it is open or closed.

In contrast to a common coding issue, a multi-project coding issue can concern multiple projects. Instead of replicating the coding issue to every service's issue management system and creating a technical debt due to these clones, it is the same coding issue for every project. This allows changes in the coding issue's body text to be recognized by all projects' developers. An example for a multi-project coding issue concerning two service projects is shown in Figure 3.4, and explained later in this section. Sometimes, if a coding issue concerns multiple projects, it could be better to separate the parts for each project to single coding issues. While this approach is easy to fulfil, the relation of these coding issues to each other is lost. Therefore, this concept extends the coding issues meta-model to provide links between coding issues. As a result, dependencies between coding issues can be represented directly. A link is not only a URL as it might be in common issue management systems. Rather, the linked coding issue is directly integrated into the actual coding issue. A developer can thus see immediately both the dependencies on another coding issue and the contents of the linked coding issue. What is special about this is that the linked coding issue does not have to belong to the same project and, therefore, does not lie in the same issue management system. An extensive description is provided under Section 3.4.2. In order to enable quality-of-services contracts, the meta-model for a multi-project coding issue contains non-functional requirements supporting the coding issue. These non-functional requirements built some constraints for the implementation that should be fulfilled in order to resolve the coding issue. An extensive description is provided under Section 3.4.3.

This thesis also introduces the concept of coding issue shadows. Since a developer of a (multi-project) project which uses the thesis' system does not necessarily have write access to imported projects within the multi-project project, some edits to coding issues through the system can't be propagated to the issue management system where the original coding issue is stored. Therefore, a shadow must be created which contains information of the original issue's body text and the changes made to it. A shadow represents a copy of the original issue containing the edits. There are several ways to handle coding issue shadows, in case the user has the relevant access rights for it. Otherwise, a coding issue shadow is only visible by the multi-project project developers. The edits made to the shadow's text body can be propagated to the original coding issue as comment, if access rights enable this. If the original issue's body text is updated, these edits can be propagated to the coding issue shadow where they can be maintained or discarded by the developers. Comments in the original coding issue can be displayed in the coding issue shadow as well. The other direction works only if necessarily rights are granted.

Multi-project coding issues have a visibility. They can be publicly visible which means they are stored in the concerning service's issue management system or private visible. In case of private visibility, a multi-project coding issue is stored in the (multi-project) project's issue management system.

For managing multi-project coding issues in a qualitative way, a concrete graphical syntax was developed, and a graph notation was conceptualized. This graph notation contains the projects/services, service interfaces and coding issues as nodes. At least one project/service can be connected to another project/service over a service interface with edges. Coding issues belonging to a project/service are pinned to the relevant project/service. If they are concerning a service interface, they can be pinned to this interface instead. To support some relation between coding issues, a multi-project coding issue can link to other coding issues, which is also shown in the graph notation. The graphical coding issue notation is described in-depth in Section 3.6.

An example use case containing a multi-project coding issue which concerns two services is provided in Figure 3.4. The figure compares how multi-project coding issues are

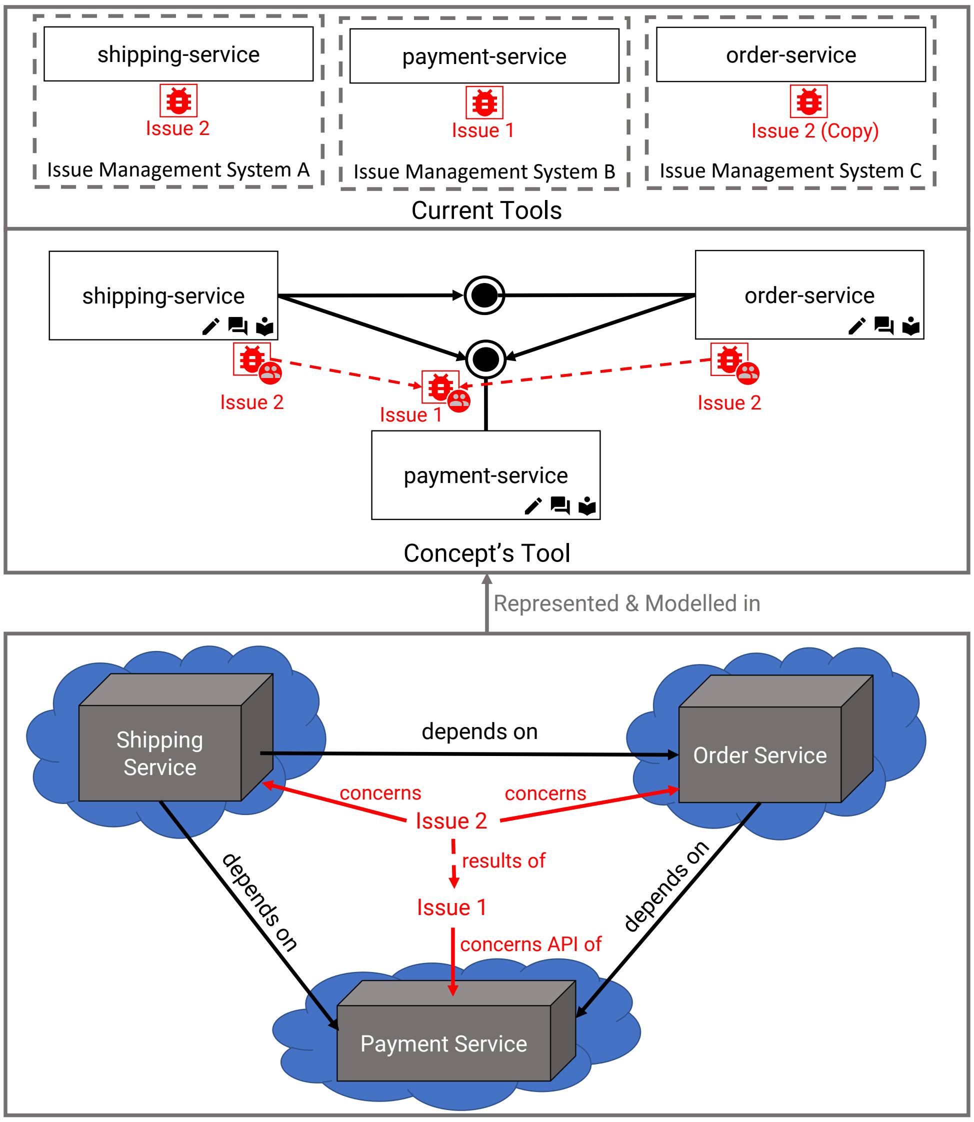


Figure 3.4: Use case example with a multi-project coding issue for two services

managed by this thesis' concept in comparison to current existing tools. It shows three services, a shipping service, an order service, and a payment service. The shipping and order services are depending on the payment service and access it through its interface. In addition to this, the shipping service depends on the order service. During development work on the payment service, an issue occurs at the interface of the service. As a result of this issue, a second issue arises which leads to the other services failure. These issues can be represented and modelled in current tools or the system's concept tool as coding issues. Using the system's concept tool, shipping service and order service have the same coding issue. It has only to be created once and is shown for both services after that. To illustrate the dependency between the coding issues, the system's tool can be used to link the causing coding issue. This link is displayed directly. If, for example, GitHub or Redmine are used to represent the issues, every service has its coding issues stored in its own issue management system. Instead of having the same multi-project coding issue for the shipping service and the order service, a copy of the multi-project coding issue must be created which leads to a clone. If one of the multi-project coding issues is edited, the edits should be done on the other too in case they concern both coding issues. Otherwise some updates are lost, and technical debt would have been created. The dependency to the causing coding issue cannot be displayed in such tools. To counteract this, the URL to the causing coding issue would have to be written into the body text of the respective multi-project coding issues. An actual linking is not possible. Since this thesis' tool would use common issue management systems in the background to manage coding issues links have to be provided as URL in body text too and additional meta-data can be added. However, the tool can abstract the URLs in its UI and is able to show the linked coding issues directly instead of a URL.

3.4.2 Linking Coding Issues to Artefacts

The second main aspect of this thesis' concept is linking multi-project coding issues to artefacts. On the one hand, linking artefacts enables better documentation of the issue. A developer does not need to describe the concrete location of the issue, he could also link it. As a result, other developers can better understand the issue. On the other hand, changes to artefacts, especially models, can be better recognized and synchronized. In addition to this, model changes can be documented in a convenient way, which also eases cross-team communication for such issues. According to the conclusions of Section 2.2.2, the artefacts should be linked as fine grained as possible. Regarding source code as an artefact, instead of complete source code files, a developer should be able to just link some lines of code, e.g. an operation. It must be noted, that the artefacts do not necessarily have to belong to the project/service the coding issue is created for. Instead, it is possible to link artefacts of other projects/services. In order to provide qualitative suggestions for artefact links, the system could retrieve structural and semantic information through NLP for text mining of models and other artefacts, as well as (syntax) analysis of source code. This information could be used to create an ontology. When creating a coding issue, the body text of the coding issue can then be analysed using NLP and the ontology can be used to suggest links to the creator of the coding issue.

Since some traditional issue management systems do not offer the ability to provide such links to artefacts, they must be parsed by the system from the body text of the coding issue. Other issue management systems, e.g. Jira, link to source code or git commits in their meta-data. However, instead of adding a copy of the source code or model to the coding issue, it must be a real link towards the actual artefact and a history in case the artefact was changed. Otherwise, changes in the linked artefact while the coding issue is open will not be recognized. As a result, the links could outdate fast. While whole files are easy to link, this becomes more difficult with even more fine-grained components. A fine-grained linking is still possible, if the artefacts are text-based, e.g. source code, as well as JSON or XML files, which represent models.

Figure 3.5 shows a sketch of a use case for linking a coding issue to an artefact. In this use case, there are two services, a shopping cart service, and an order service. This shopping cart service depends on the order service. Each service is developed by its own developer team. For a new release, the order service's developer team updates the service's interface and an interface description model. The developer team of the shopping cart service wants to update their service as well in order to use the new release of the order service. Therefore, a developer of this developer team creates a feature request for the shopping cart service to document the task. He links the part of the order service's interface description model and the order service's request endpoints of the source code which must been changed in the new release. As a result, it is clear for all developers of the shopping cart service's developer team which changes they must implement.

3.4.3 Non-Functional Requirements in Coding Issues

As stated in Section 2.2.2, Capilla et al. outline that quality-of-service conditions and other run-time decisions should be supported in design decisions as well [CZZ+11] which are modelled as coding issues here. Therefore, adding non-functional requirements and constraints to coding issues is the third main aspect of this thesis' concept. These non-functional requirements and constraints can be checked with some extension tools, e.g. use of monitoring systems like Kieker. For example, latency limits for requests to interfaces can be specified and checked by monitoring systems. If the limit is not satisfied, the corresponding pull request can be rejected automatically. This approach is similar to the WS-Policy properties from the WS-* world and therefore continues the idea that a contract between the parties involved, in this case the coding issue and the processing developers, must be fulfilled. Thus, a specific set of assertions must be agreed upon that will describe each feature. If these requirements are extensible, it allows for custom non-functional requirements to be created as well. Therefore, the system must provide a meta-model for custom non-functional requirements and constraints. If possible, the developer should be able to link his own checks and monitoring systems to the requirements, thus improving developer productivity when solving coding issues. The check of requirements can not only be for acknowledgement purpose. Rather, it

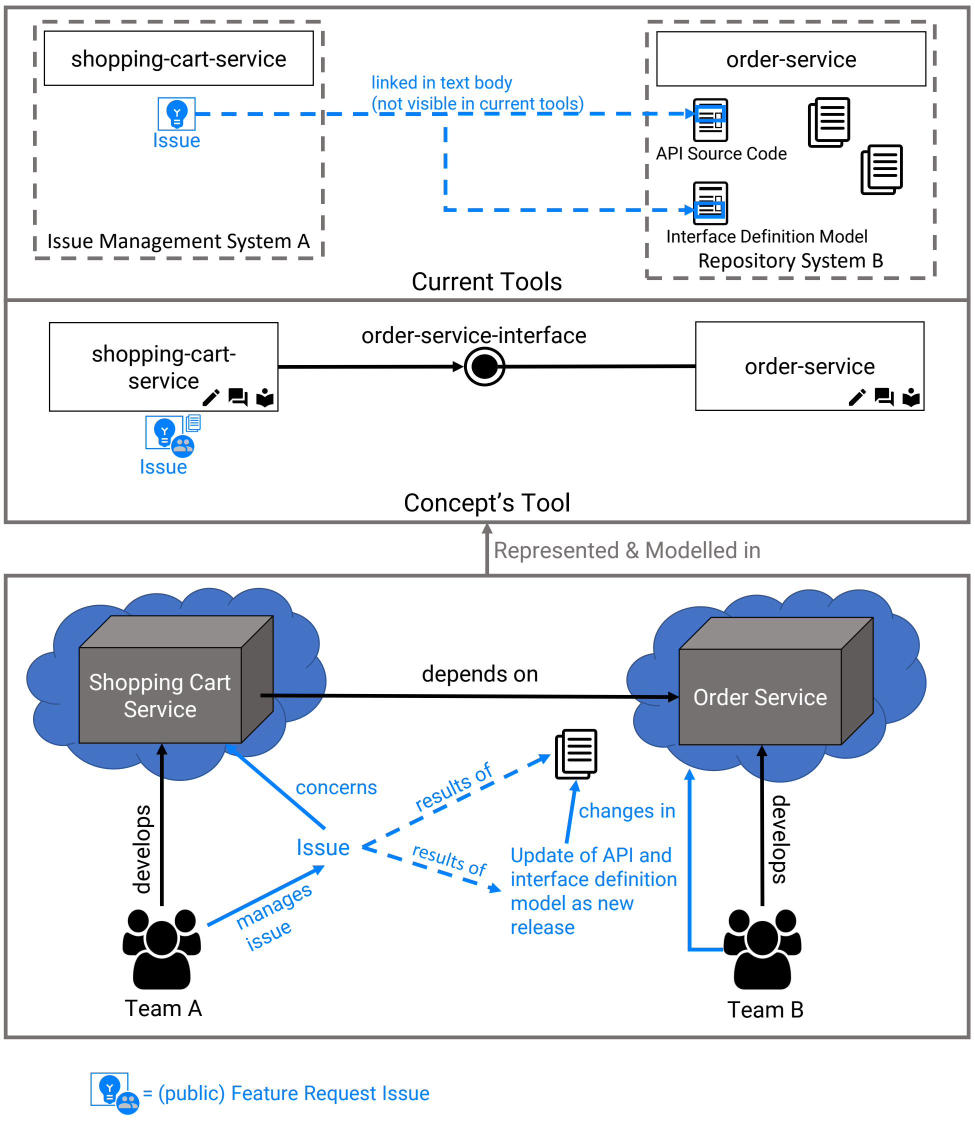


Figure 3.5: Use case sketch for linking coding issues to an artefact

can be possible to automatically reject pull requests for failed checks, as with a CI/CD. This quality assurance measure prevents bad source code from being maintained in a production environment. As a result, quality-of-service should be improved.

As with artefact links, traditional issue management systems do not provide the ability to specify non-functional properties and constraints. For this reason, they must also be parsed in the text body of a coding issues. In order to be able to check these non-functional properties and constraints, the system of this thesis should offer the developer a possibility to select appropriate checks and execute them as automatically as possible. In addition, it would be useful to execute the specified checks as soon as a pull request

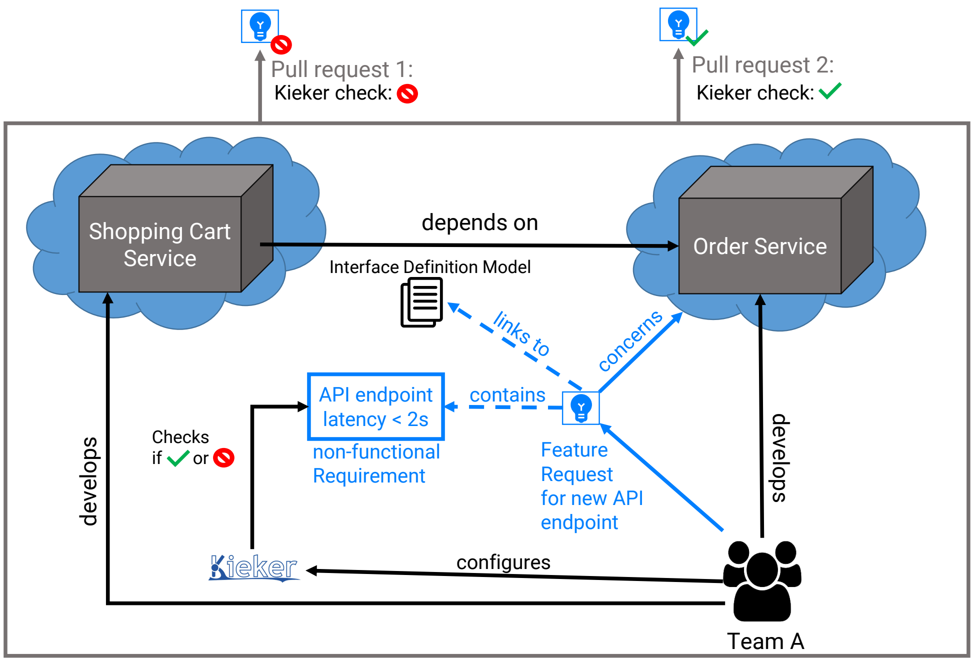


Figure 3.6: Use case sketch for a non-functional constraint in a coding issue

with the corresponding coding issue has been completed. It should only be possible to accept the pull request if the checks have been run successfully. If the non-functional constraints were set too high, they may have to be adjusted. In this way, at least the documentation of the coding issue can be kept up to date.

A use case example for a non-functional constraint is shown in Figure 3.6. There is shopping cart service which is connected to an order service. Both services are developed by the same developer team. During the development process, a new endpoint for the order service has to be implemented. Therefore, the developer team creates a feature request describing the endpoint. In addition to this, they add the interface definition model to the coding issue. To ensure good latency for processing a request to the endpoint, developers add a non-functional constraint to the coding issue. As a constraint, they specify that the processing of a request to the end point can only take an average of 2 seconds. To check this, the developers configure the monitoring system Kieker [HWH12], with which they can measure requests. During their first implementation, the developers notice that the requests take too long. The check of the constraint is therefore not successful. For this reason, they still need to implement some performance improvements. The second implementation, on the other hand, fulfils the constraints and the pull request can be approved.

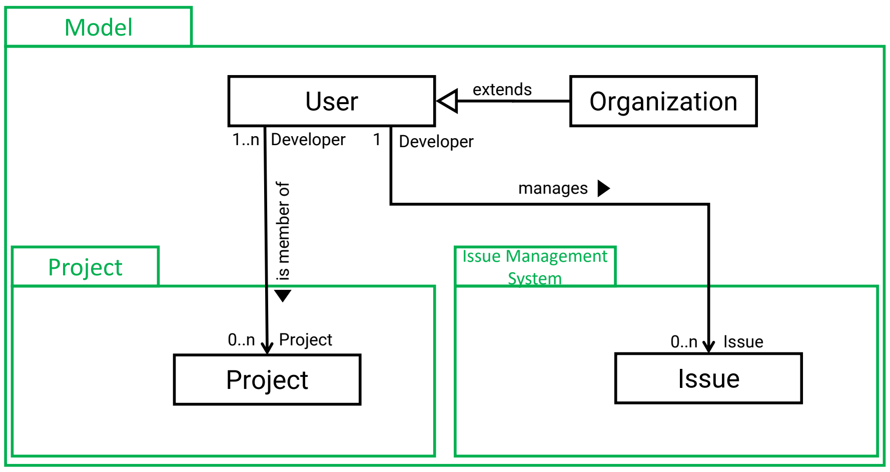


Figure 3.7: The model package

3.5 Domain Model as UML

Resulting from the gathered requirements from an initial expert survey, and scenarios which were described in Section 3.2, a domain model for an multi-project coding issue management system was created. In order to provide a better overview, the domain model is divided into sub-areas and these are presented individually in the following.

The first part is from the point of view of a user of a multi-project coding issue management system. Figure 3.7 shows the main model package of the domain model. It contains the user, a project sub-package and an issue management system sub-package. The project sub-package contains the domain part of the project, while the issue management system sub-package contains the domain part for managing multi-project coding issues. While a user as a developer can be a member of none, one or more projects, a project contains as many as you like, but at least one developer. Each of these projects corresponds to a multi-project project, i.e. a project that consists of several (independent) sub-projects that are services at the lowest level. A developer can manage any number of coding issues. Each coding issue, on the other hand, has exactly one creator.

The part of the domain model regarding a multi-project project is managed in the project package which is depicted in Figure 3.8. Every (multi-project) project has exactly one developer user or organization as an owner and consists of one or more components. A component is representing a sub-project or service. These components can be composed of other components. Every project has one project component which is the main component to store and manage project meta-data and private coding issues. Each component manages its coding issues in an issue management system, which can be

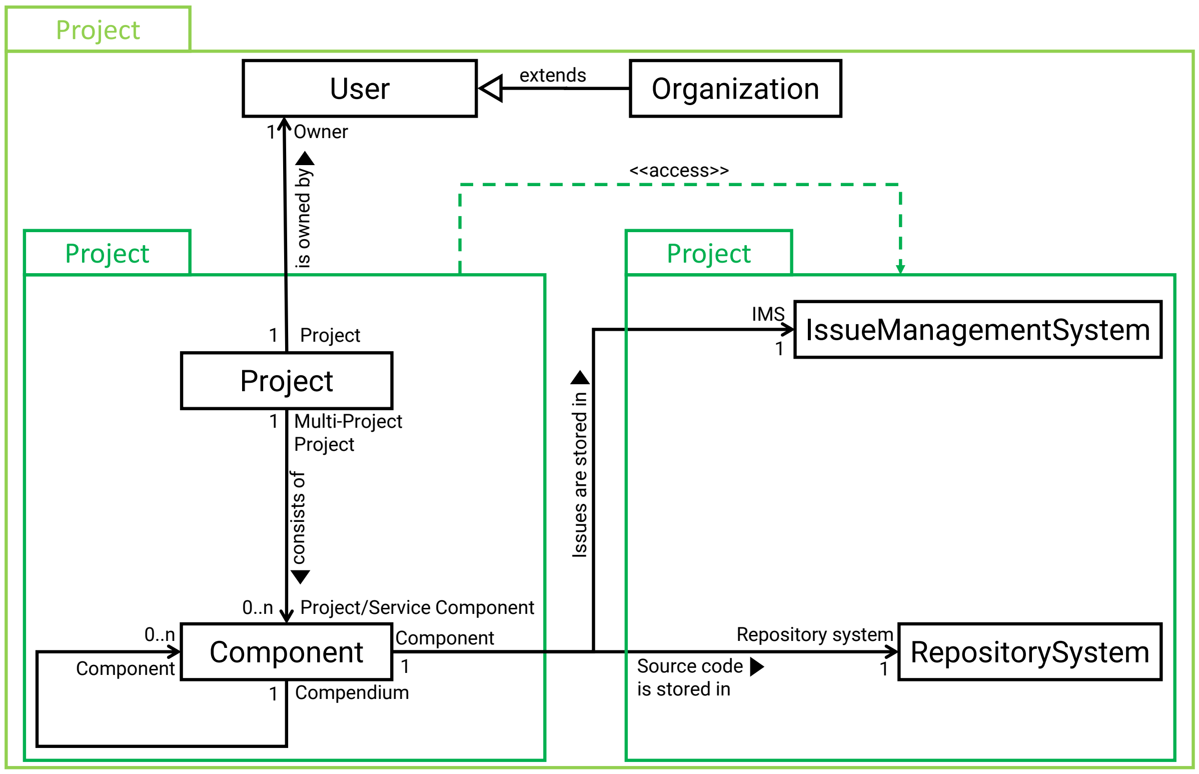


Figure 3.8: Project part of the domain model

hosted on GitHub, GitLab, Redmine, Jira or other providers. Coding issues can be multi-project coding issues or single-project coding issues. Coding issue's shadows and all private coding issues are stored in the project component's issue management system. The source code of each component is stored in a repository system.

The issue management package of the domain model is shown in Figure 3.9. It describes the relation between an issue management system, a repository system, and a coding issue. An issue management system can be a normal issue management system or a multi-project coding issue management system. Every issue management system, as well as every repository system, is hosted by one provider. Such a provider for an issue management system can be for example GitHub or Jira. For a repository system a provider can be for example GitLab or GitHub. Coding issues are describing problems or features of the source code stored in a repository system. They are stored in an issue management system which means every repository system holds its coding issues in an issue management system. A coding issue can be a multi-project coding issue, whose meta-model is already shown in Figure 3.3. As a result, every coding issue can have multiple meta-data which can be linked to multiple additional artefacts. These artefacts are models or source code stored in the repository system's code repository.

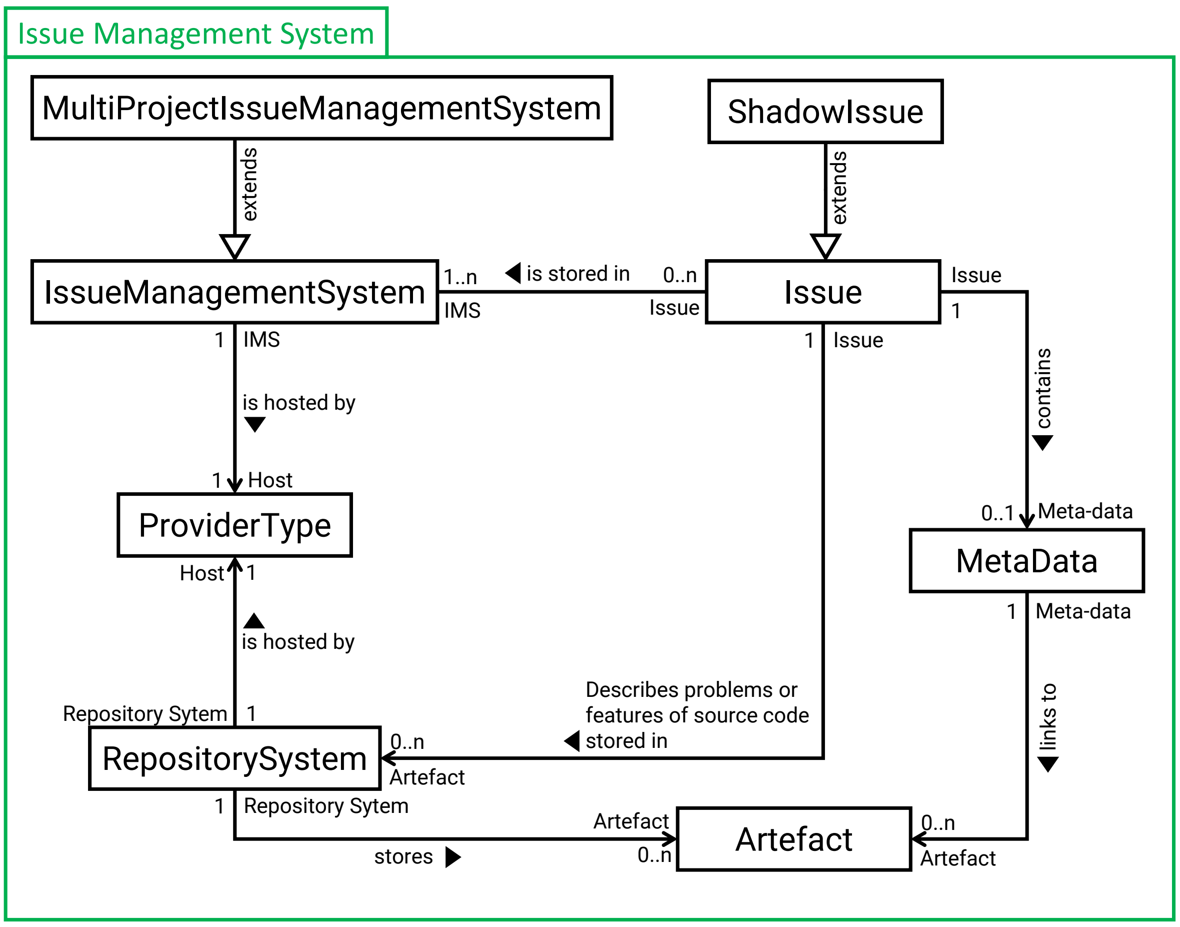


Figure 3.9: Issue management model

3.6 Multi-Project Issue Modelling and Notation (MPIMLAN)

To qualitatively manage multi-project coding issues and their concerning projects/services in a multi-project coding issue management system, a modelling language was created, the Multi-Project Issue Modelling and Notation (MPIMLAN). The idea of this concrete graphical syntax is to manage multi-project coding issues in a service dependency graph notation as depicted in Figure 3.10. The graph notation contains an unconnected bipartite graph representing the services and their connection to each other. In addition to this, the graph notation contains multi-project coding issues as additional nodes which can be linked to each other.

The unconnected bipartite graph has two types of nodes, services, and service interfaces. Services can be connected directly or indirectly through service interfaces to each other. While the boxes represent service nodes, the service interface nodes are represented by

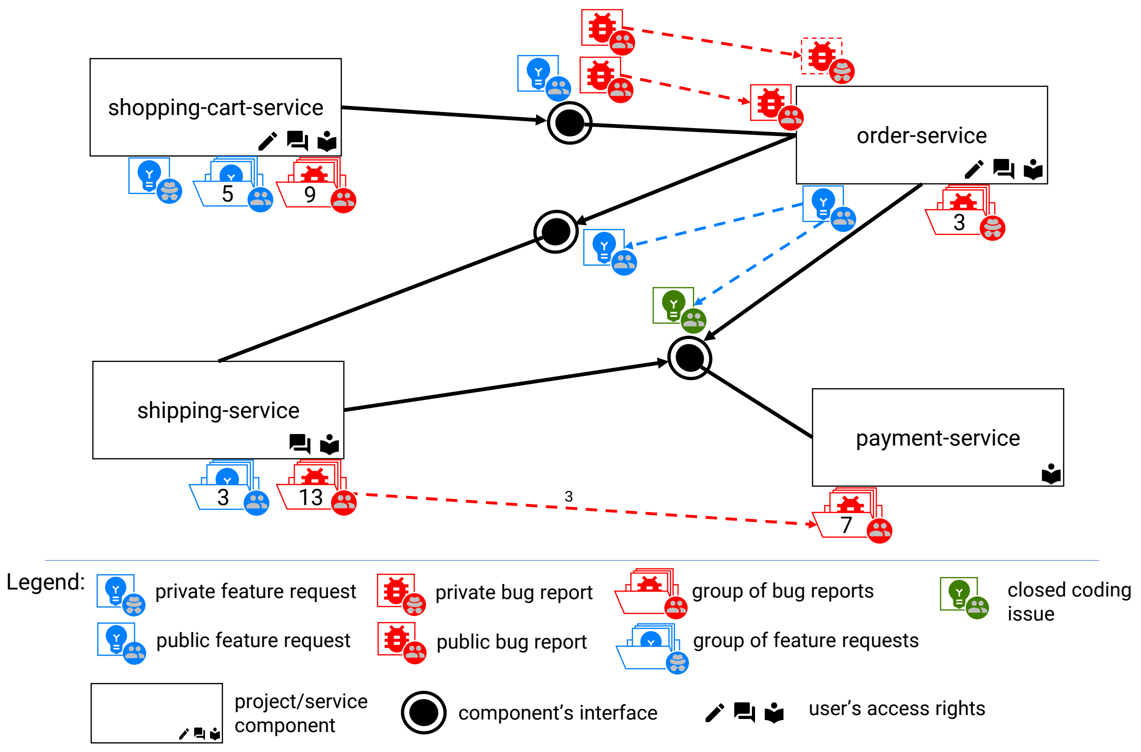


Figure 3.10: Multi-project Issue Modelling and Notation

circles with filled out dot in the middle. In service interface nodes incoming edges end at the edge of the outer circle. Outgoing edges, on the other hand, start in the inner circle of the node. A service can have multiple interfaces which means it can be connected to several service interface nodes in the graph notation. Every service node contains the access rights of the current user of the multi-project coding issue management system. A user can have write access, comment access or read access, most likely combined.

Each service can have several coding issues, which are represented as multi-project coding issue nodes in direct surrounding of the service node. However, if a coding issue refers to a service interface it is displayed in the direct environment of the service interface node. Coding issues can be feature requests or bug reports. While feature requests are represented in blue with a lamp icon, bug reports are represented in red with a bug icon. Every coding issue node shows the visibility of its issue. Private coding issues have spy icon attached while public coding issues have a group icon attached. In addition to this, a multi-project coding issue can link other coding issues as described in Section 3.4.2. These links are also represented in the graph notation. Usually closed coding issues are not depicted in the graph notation. However, if a closed coding issue is linked by a multi-project coding issue, it is shown green instead of its original colour. In order to ensure clarity in the graph notation, coding issues of the same type, e.g. if the type of coding issue and the visibility are the same, are grouped together and represented as a folder. The folder contains the number of coding issues. Also, the links between coding issues can be combined. If, for example, three multi-project coding issues, a grouping with 13 coding issues, link with three coding issues of another grouping with 7 coding issues, only one link with a counter of 3 is displayed between the groupings. The graph notation can show coding issue shadows as well. They are illustrated with dashed lines instead of solid lines around the node.

Chapter 4

Implementation

This chapter described the implementation of this thesis' concept. In Section 4.1 an overview of the main features the implementation concentrates on, and the implementation's architecture are presented. Afterwards, Section 4.2 back-end and front-end of an implementation prototype are outlined. This section shows UI mock-ups to imagine how a system looks like using the Multi-Project Issue Modelling and Notation (cf. Section 3.6). Finally, in the prototype's implementation used tools and technologies are elucidated in Section 4.3.

4.1 Overview of the Main Features and Implementation's Architecture

This section describes the main features the concept's implementation prototype concentrates on and important components. Afterwards, the implementation's architecture is sketched and described. It will be stated out, which components must work together in order to achieve a minimal example.

The main objective of the prototype implementation of the concept is to verify the feasibility of the concept itself. Therefore, the prototype focuses primarily on creating a breakthrough of the functionality instead of covering the complete functionality. As the focus of the implementation, the prototype concentrates on recursively hierarchically structured projects and coding issues that affect single or multiple projects. The implementation should enable a developer to create, edit, comment and close multi-project coding issues. A multi-project coding issue should be able to link other coding issues. Furthermore, other developers should be able to be assigned to the coding issue. For the sake of simplicity, labels should first be provided as strings to the coding issue, if they already exist in all involved issue management systems. In order to get a quick

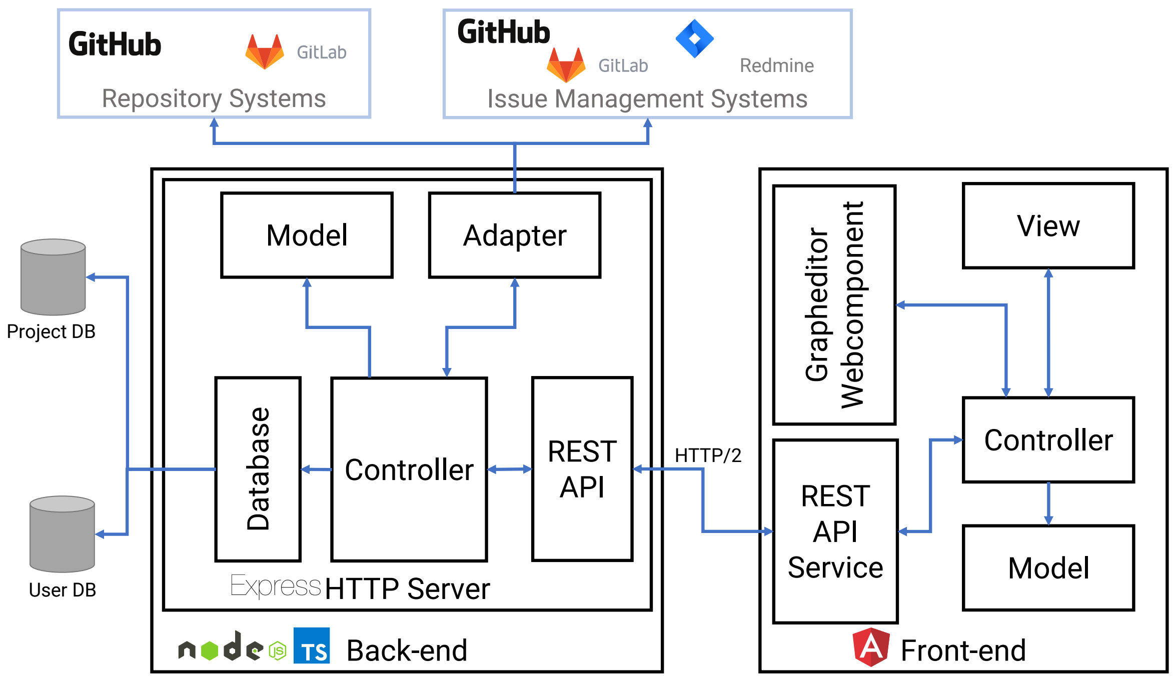


Figure 4.1: Prototype implementation's architecture

prove-of-concept, the first prototype contains only the linking of whole files as artefacts. A finer-granular linking can be delivered afterwards in a second version of the prototype. In addition, the Multi-project Coding Issue contains non-functional properties for documentation purposes only. An (automatic) check via monitoring systems will only take place in a future version. To make this possible, however, pull requests via the system must first be made possible so that the checks can be started, and the feedback can be used in a meaningful place. Furthermore, the first prototype contains only a basic support for multi-project coding issue shadows. These can be generated when editing a multi-project coding issue. When querying the coding issues, the original coding issues are filtered out using a comparison. An important main feature is the graph editor, which forms the heart of the prototype. It should be able to display and filter the coding issues and components. The concept's implementation tries to stay as modular as possible. Therefore, adapters are used to communicate with the issue management systems and repository systems. Through the adapter architecture, the system can be extended easily if a new issue management system or repository system should be provided. The API logic must be separated from the business logic in order to gain a clearer control flow.

Based on these main features and considering the remaining requirements from Section 3.2, an architecture for the prototype was developed. This architecture can be seen in Figure 4.1 and is described in the following. The back-end is divided into four main packages: API, adapter, model and database. It is described below. The front-end's archi-

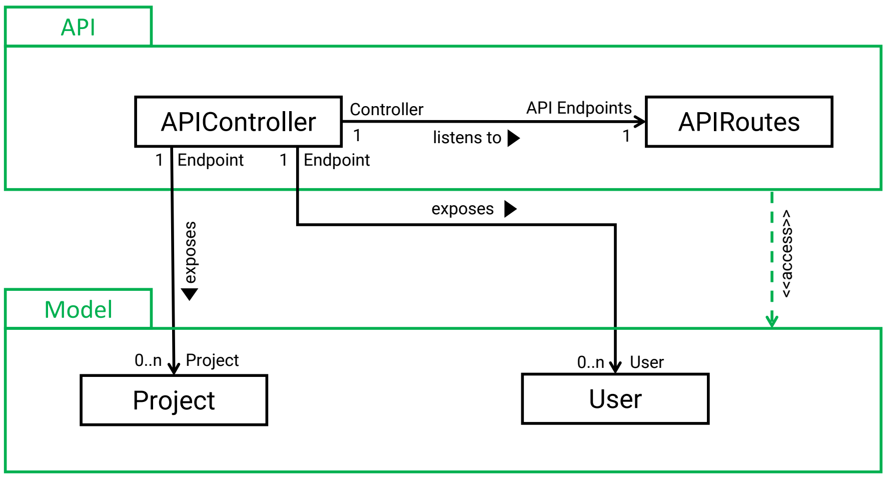


Figure 4.2: API package of the architecture

tecture is based on a standard MVC pattern and built with Angular's project structure. Back-end and front-end are communicating via HTTP requests with each other. The model package of the back-end contains the classes defined in the domain model in Section 3.5. Therefore, it will not be described here again.

The API consists of an APIController and an APIRoutes class, as shown in Figure 4.2. In the APIRoutes class the API routes to the endpoints are defined and the listeners are implemented. Additionally, middlewares are defined for validation and authentication of the requests that are called during a request. The actual implementation of the logic of an endpoint is defined in the APIController class. The functions implemented there are called after all specified middle-ware at the end of an endpoint. This allows a clean separation of endpoint and controller. In the controller, the steps of the endpoint to be completed are carried out using the model classes. If any data from the database is needed, the API package gathers it from the database package.

The database stores data that is not a multi-project coding issue or can't be stored in the project's repository system. This data is managed by the database package, shown in Figure 4.3. There are two controllers, a UserDBController and a ProjectDBController. User credentials are managed by the UserDBController, while the multi-project projects and their (sub-)components are managed by the ProjectDBController.

The last main package contains the issue management adapters and repository system adapters in sub-packages. Each kind of adapters has a father class for generic abstraction of the concrete adapter. Therefore, the concrete adapter can be determined and instantiated at runtime. The adapter sub-packages with their classes can be seen in Figure 4.4 and Figure 4.5.

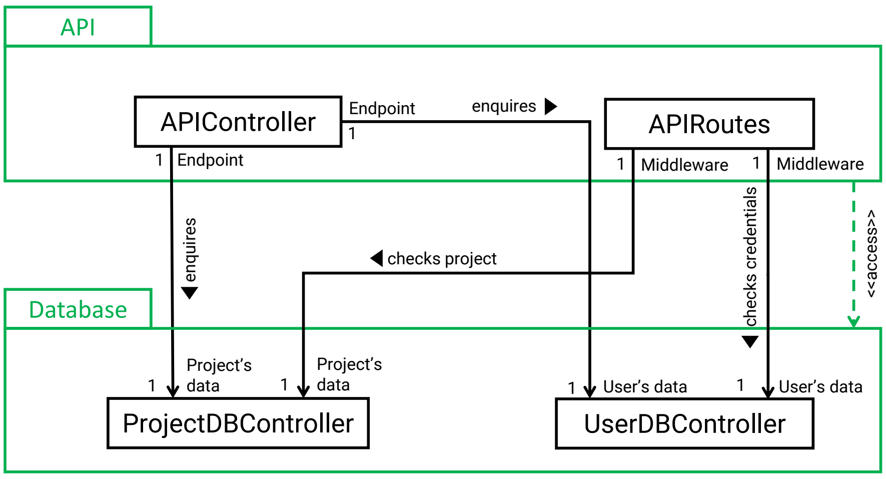


Figure 4.3: Database package of the architecture

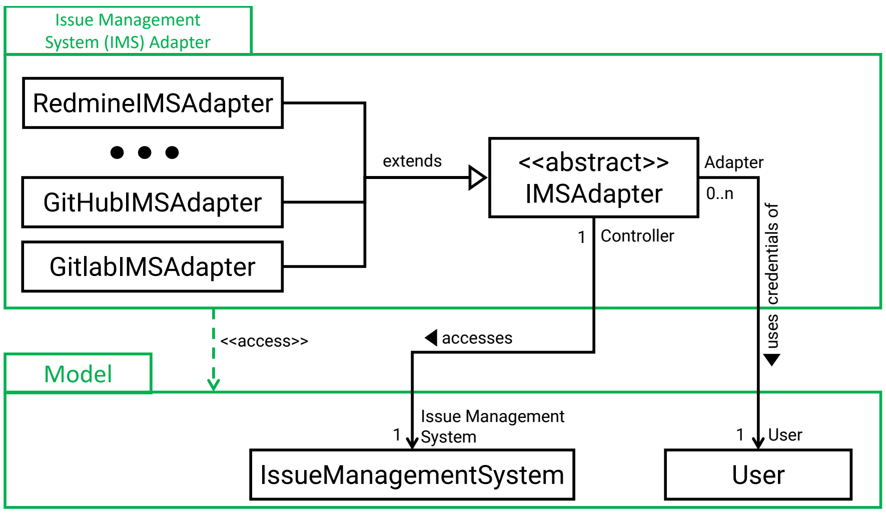


Figure 4.4: Issue Management System Adapter sub-package

In order to achieve a minimal working example all back-end components should be implemented. To interact with an issue management system and a repository system at least one adapter for each kind of adapter is needed. Regarding the front-end, components to create multi-project projects, the Grapheditor Webcomponent component, some functionality to manage multi-project coding issues and components to communicate to the back-end have to be implemented.

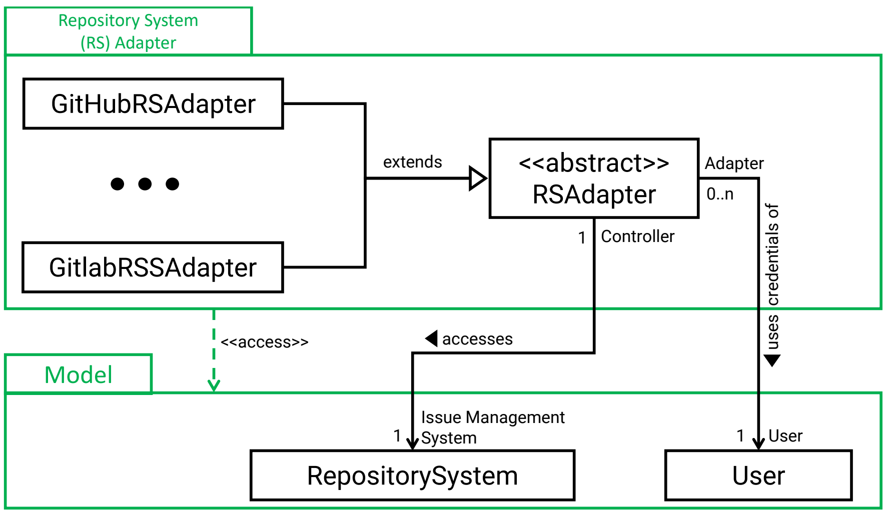


Figure 4.5: Repository System Adapter sub-package

4.2 Prototype Implementation

This section describes the prototypes implementation. First, the back-end is described. Afterwards, information about the front-end is provided. It has to be noted, that both, back-end and front-end could not be completed in the time window of the master's thesis. To compensate this, mock-ups for the front-end were created. These mock-ups are shown in this section as well and were used for the evaluation described in Section 5.3. The source code of both the back-end and the front-end can be found in the repositories of the MPMTI GitHub organization[1].

The back-end holds the prototypes business logic. The prototype allows a user to create multi-project projects. The sub-projects of a multi-project project are represented as components in the implementation. Since the projects are recursively hierarchically arranged, each component contains a list of subcomponents (sub-projects), like a node list of a graph. Each component is assigned to an Issue Management System and a Repository System. For a first breakthrough, the implementation focuses on GitHub as a provider for both an issue management system and a repository system. Therefore, only adapters for GitHub exist yet. The prototype supports now just a basic implementation of coding issue shadows without any synchronization to the original coding issue. When querying the coding issues, the shadows are identified by the coding issue title and the name of the originator and the original coding issues are filtered. To build the graph components in the front-end, the back-end can iterate through the components of a multi-project project and return a JSON object to the front-end that has all the relevant

[1]https://github.com/mpmti

content. The circular structure of the graph cannot be mapped directly to the JSON object. For this purpose, each component receives a list of dependent components in addition to its children.

The front-end's implementation can only create multi-project projects, manage them and retrieve the (sub-)component them right now. All other features are mocked-up yet. However, the dialogue mock-up source codes and the view to create a multi-project coding issue mock-up are already implemented in Angular. These must therefore still be integrated into the front-end project and built dynamically. The system's mock-ups are shown and described in the following. Figure 4.6 shows the system architecture graph editor where the microservice architecture of the multi-project project and their coding issues can be seen. This system architecture graph editor is based on the modelling language introduced in Section 3.6. The figure shows an example architecture containing several coding issues. Above the graph editor are slide toggle controls placed which allow a user to filter for feature requests (the blue lamp), bug reports (the red bug) or unclassified coding issues (the question mark). In addition to this, there is a filter slide toggle control to only show coding issues with notifications which can be seen in Figure 4.8 for feature request coding issues. The puzzle button on the upper right part above the graph editor opens a dialogue to create a new component. Graph editor, toggle controls and the button to create a new component are embedded in a tab view which can be seen above them. The second tab allows a user to switch to the list view instead of the graph editor view which can be seen in Figure 4.7. However, the list view shows only components and their coding issues, not their dependencies or interfaces. On the left side of the view is a sidenav bar showing the user's multi-project projects. Pressing the plus button above allows to create a new multi-project project.

If a user clicks in the graph editor view or the list view on a coding issue a dialogue is opened which contains all relevant information for the coding issue. Such an example dialogue is shown in Figure 4.9. The dialogue allows to add assignees, labels, enable or disable notifications and close or comment the coding issue too.

Clicking in the graph editor on a coding issue group (the folder ones) opens a dialogue with a list of coding issues of the group. Afterwards, the user can choose one of these coding issues by selecting it. The list shows only the coding issues' titles. Figure 4.10 shows such a dialogue for a group of feature request coding issues. If a user clicks in the graph editor view or list view on a component a dialogue containing all information of the component is opened. Figure 4.11 shows such a dialogue for an exemplary order-service component. This dialogue allows a user to add new bug reports, feature requests or component's interfaces. A similar dialogue is shown for component's interfaces if a user clicks on one. Figure 4.12 depicts a dialogue for a service interface information.

Figure 4.13, Figure 4.14, and Figure 4.15 show parts of a dialogue to create a new component. Figure 4.16 shows the view to create a new coding issue. A user must

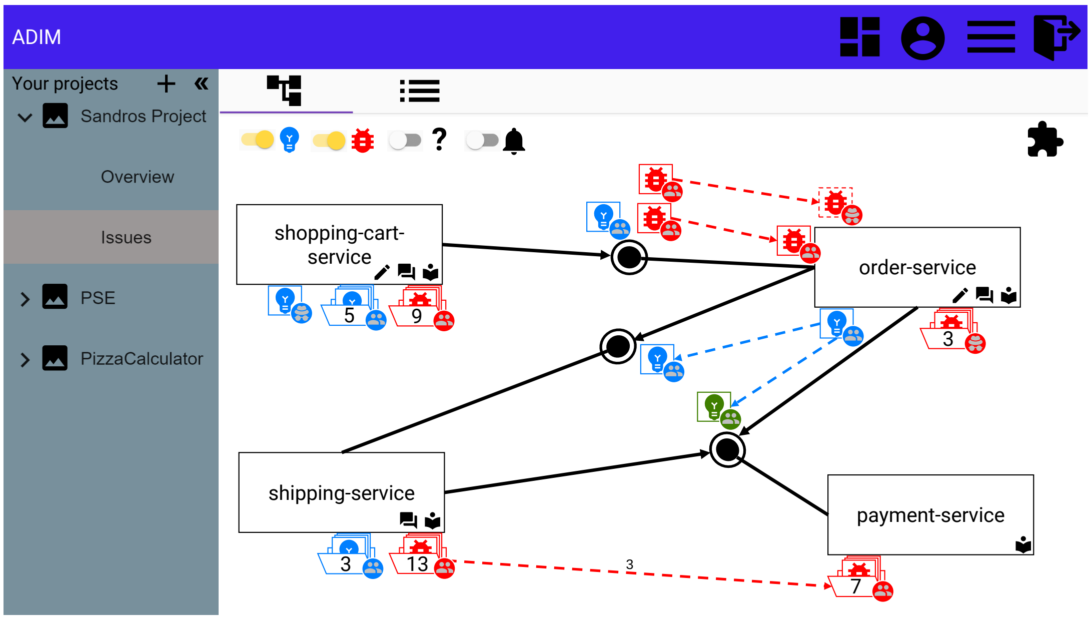
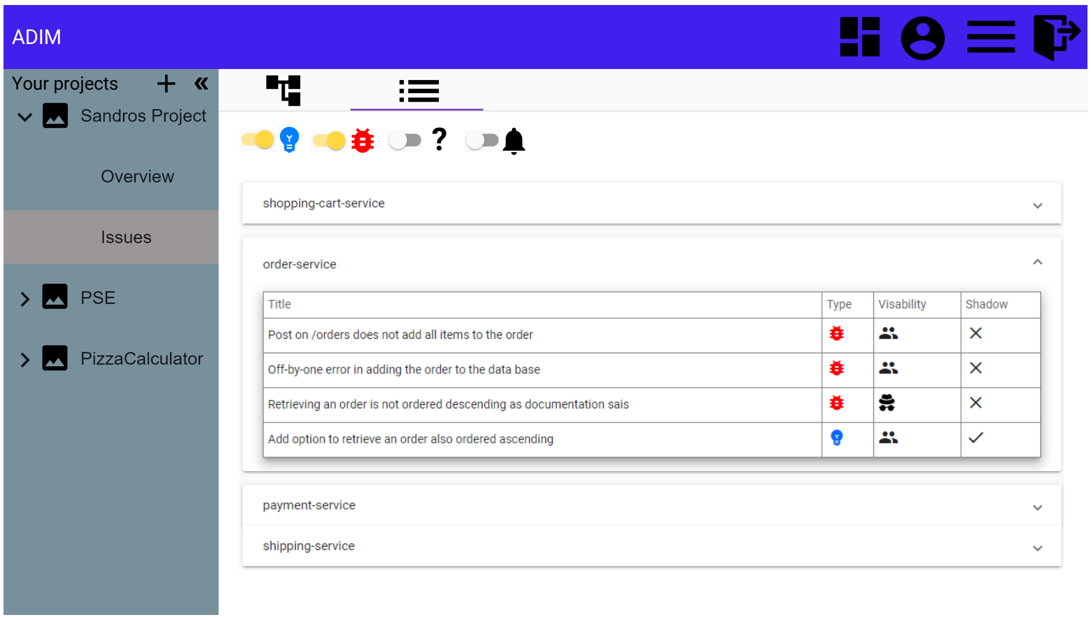


Figure 4.6: System's system architecture graph editor view with feature requests and bug report coding issues



Title	Type	Visability	Shadow
Post on /orders does not add all items to the order	🐞	👥	✕
Off-by-one error in adding the order to the data base	🐞	👥	✕
Retrieving an order is not ordered descending as documentation sais	🐞	🕵	✕
Add option to retrieve an order also ordered ascending	💡	👥	✓

Figure 4.7: System's list view of components and their coding issues

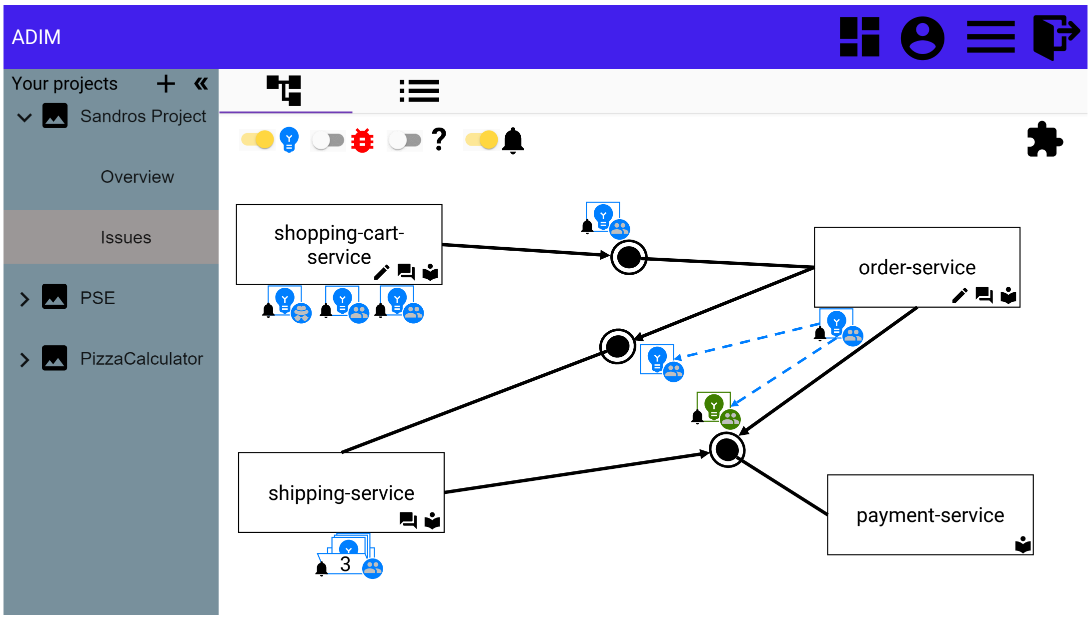
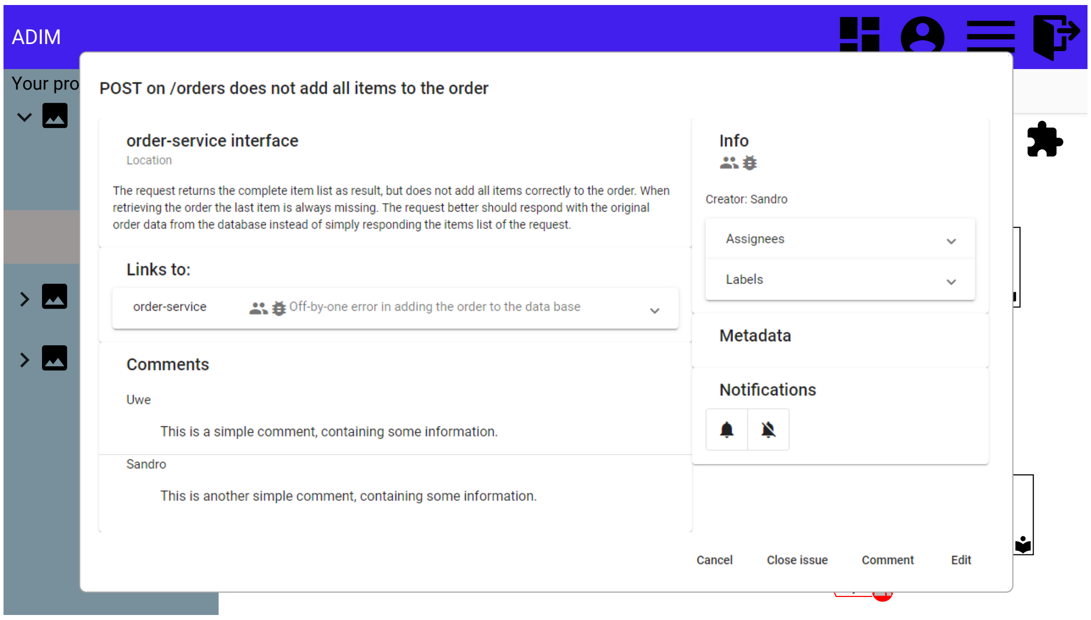


Figure 4.8: System's system architecture graph editor view filtered for feature requests with notifications



POST on /orders does not add all items to the order

order-service interface
Location

The request returns the complete item list as result, but does not add all items correctly to the order. When retrieving the order the last item is always missing. The request better should respond with the original order data from the database instead of simply responding the items list of the request.

Links to:

order-service Off-by-one error in adding the order to the data base

Comments

Uwe

This is a simple comment, containing some information.

Sandro

This is another simple comment, containing some information.

Info

Creator: Sandro

Assignees

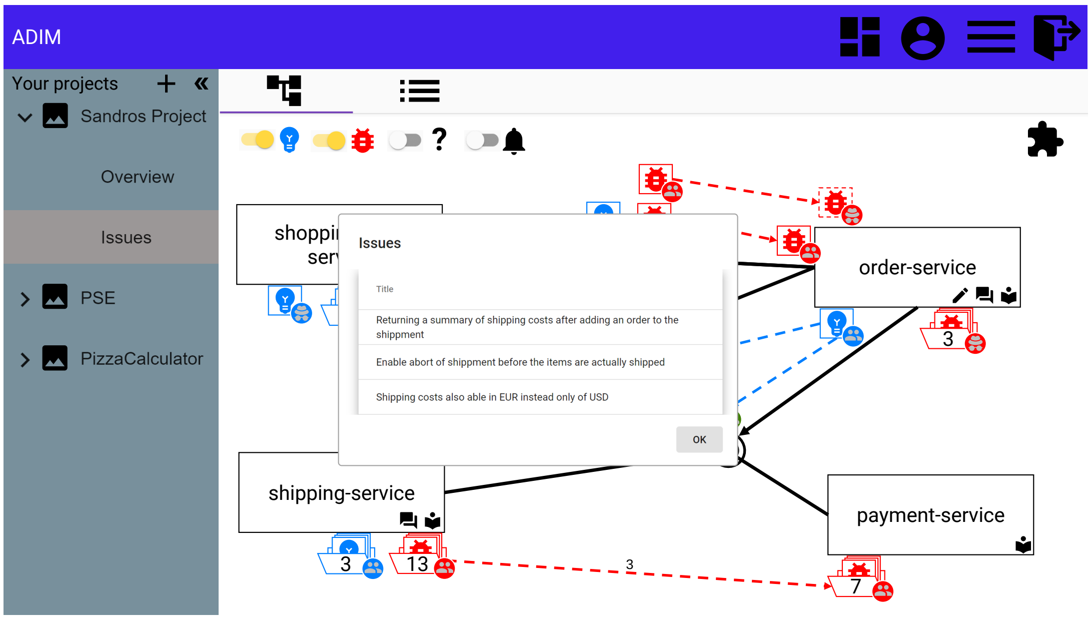
Labels

Metadata

Notifications

Cancel Close issue Comment Edit

Figure 4.9: Dialogue with information of a coding issue



Issues

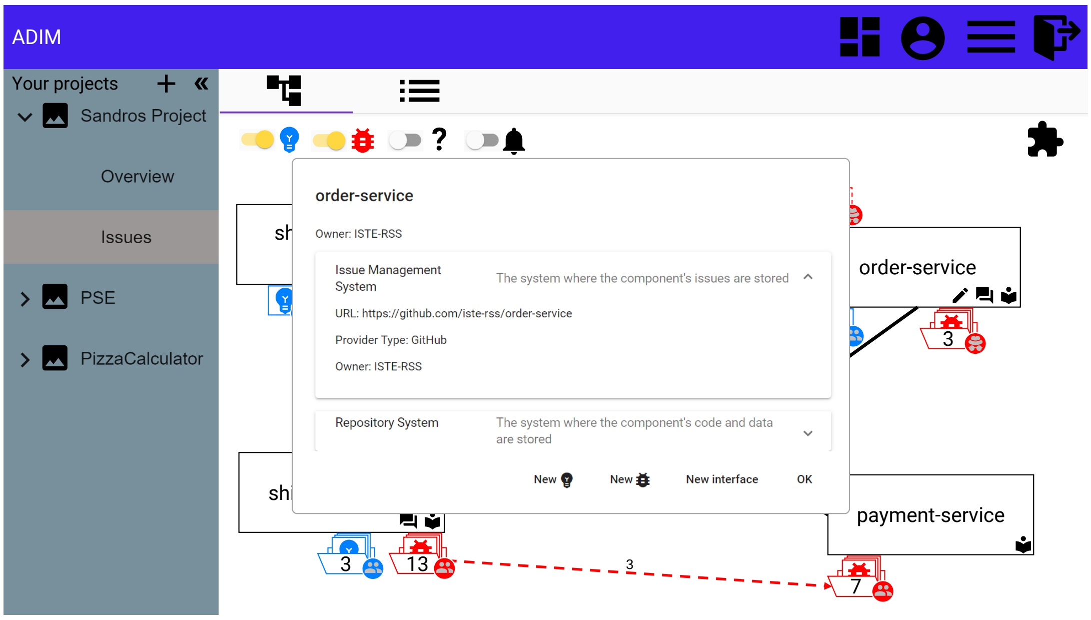
Title

Returning a summary of shipping costs after adding an order to the shippment

Enable abort of shippment before the items are actually shipped

Shipping costs also able in EUR instead only of USD

Figure 4.10: List of feature requests of a coding issue group



order-service

Owner: ISTE-RSS

Issue Management System — The system where the component's issues are stored

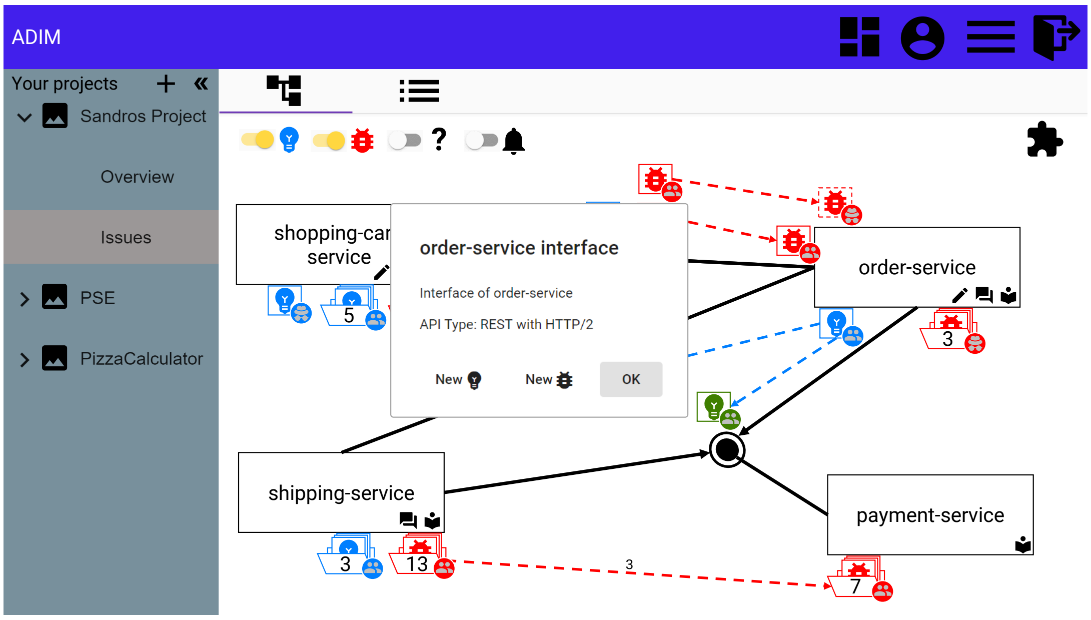
URL: https://github.com/iste-rss/order-service

Provider Type: GitHub

Owner: ISTE-RSS

Repository System — The system where the component's code and data are stored

Figure 4.11: Dialogue with information of a component



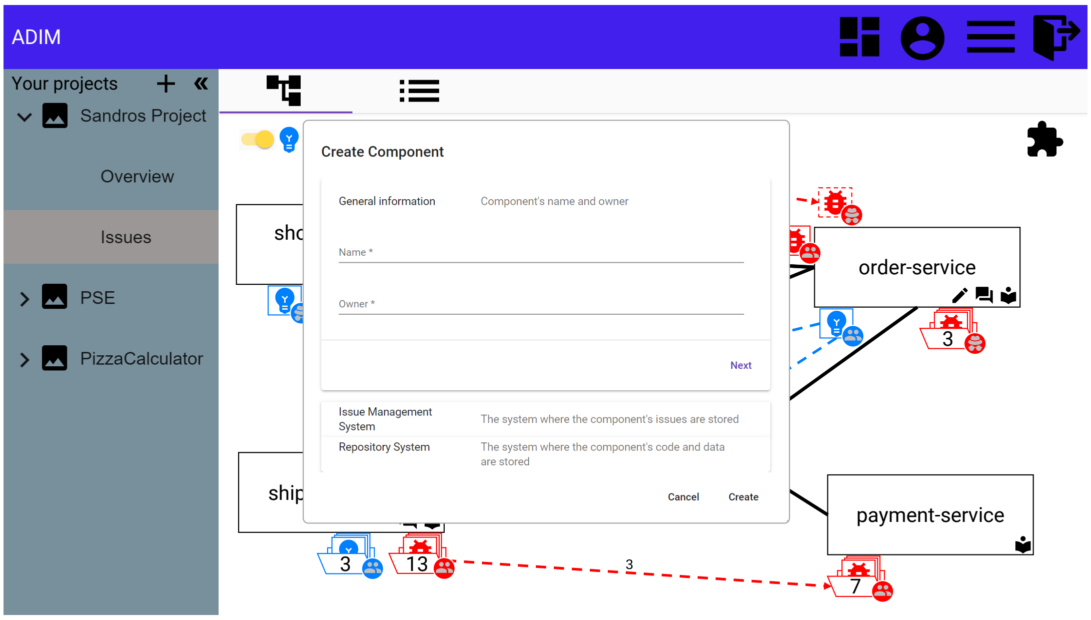
order-service interface

Interface of order-service

API Type: REST with HTTP/2

New | New | OK

Figure 4.12: Dialogue with information of a component's interface



Create Component

General information | Component's name and owner

Name *

Owner *

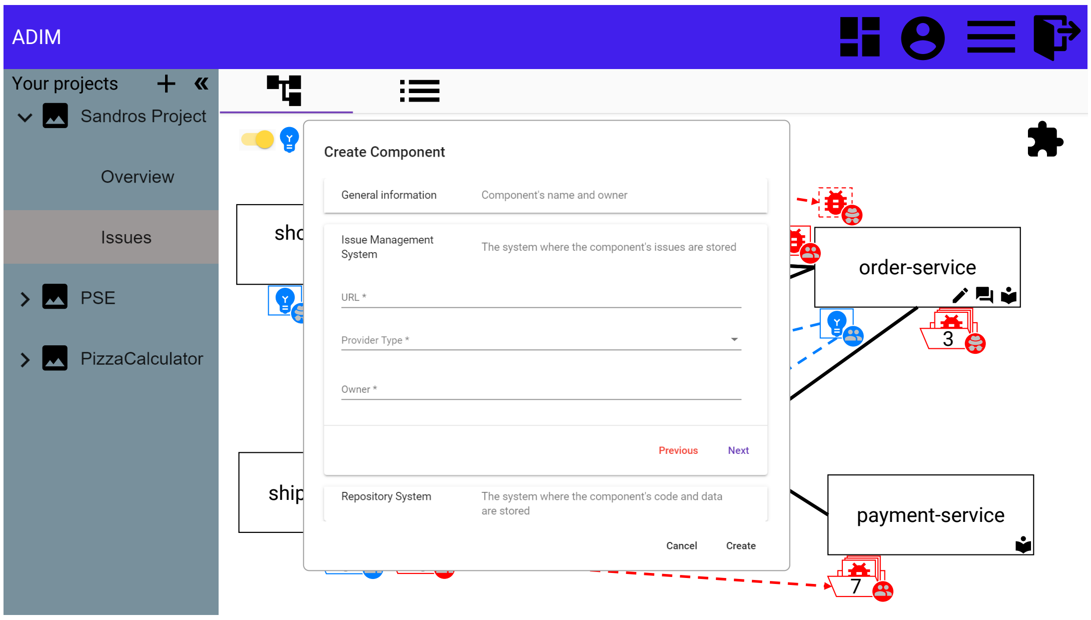
Next

Issue Management System | The system where the component's issues are stored

Repository System | The system where the component's code and data are stored

Cancel | Create

Figure 4.13: First part of a dialogue to create a new component



Create Component

General information — Component's name and owner

Issue Management System — The system where the component's issues are stored

URL *

Provider Type *

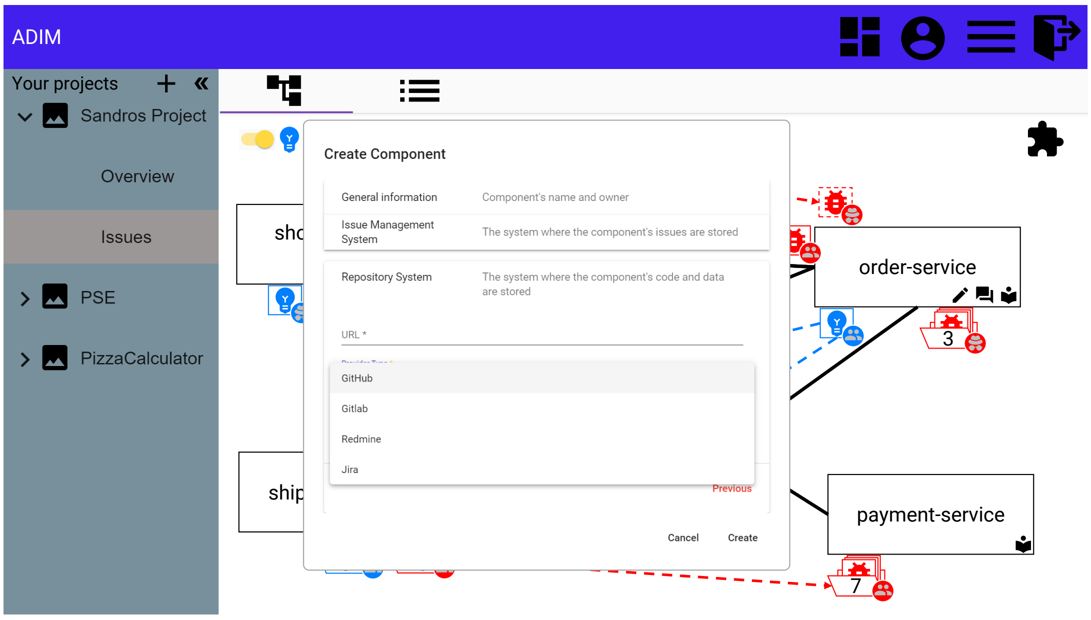
Owner *

Previous Next

Repository System — The system where the component's code and data are stored

Cancel Create

Figure 4.14: Second part of a dialogue to create a new component



Create Component

General information — Component's name and owner

Issue Management System — The system where the component's issues are stored

Repository System — The system where the component's code and data are stored

URL *

GitHub
Gitlab
Redmine
Jira

Previous

Cancel Create

Figure 4.15: Third part of a dialogue to create a new component

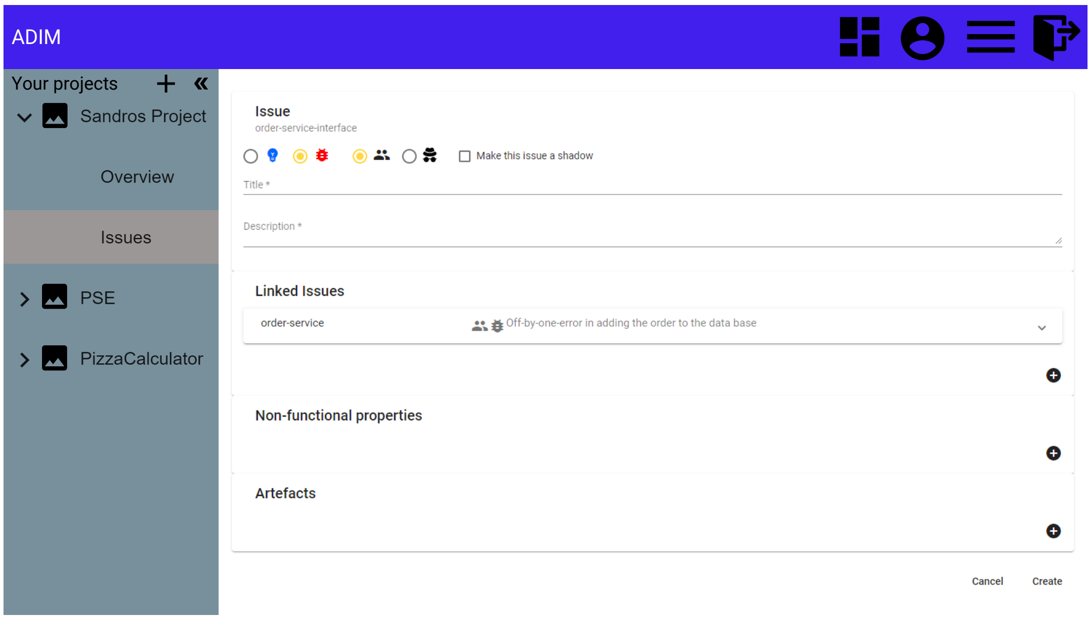


Figure 4.16: System's view to create a new coding issue

decide whether the new coding issue is a feature request or a bug report, public or private and whether it should be created as a shadow. Additionally, other coding issues, artefacts or non-functional properties can be added as links.

The graph editor itself already exists and can be included as external dependency. It is highly configurable, so creating the graph editor from the Multi-Project Coding Issue Management and Notation (cf. Section 3.6) should be possible in a few hundred lines of code.

4.3 Used Tools and Technologies

This section describes which tools and technologies were used for the implementation prototype. First, the set-up of the programming language used for the back-ends implementation is explained. Additionally, this section describes which libraries are utilized. Afterwards, the framework used to implement the front-end is described. Finally, this section describes the graph editor library which is planned to be used in the front-end to implement the graph editor of the Multi-Project Issue Modelling and Notation.

As technology for the back-end, the asynchronous event-driven JavaScript runtime Node.js[2] which is designed to build scalable applications is used. In contrary to other programming languages, Node.js's event loop isn't blocking which is suitable for scalability. Especially for HTTP servers, Node.js's design comes with a streaming and low-latency approach. Since Node.js provides performance since is built against modern versions of V8, which is written in C++. Furthermore, it implements modern ECMAScript specifications to allow a developer to work with classes and other object-oriented concepts. However, Node.js itself only provides dynamic typing as most JavaScript frameworks do. In order to gain more maintainability, TypeScript[3] is configured as addition to Node.js. As a result, the back-end project was written in Node.js using TypeScript for application-scale development. TypeScript provides strong static typing, inheritance and other supporting features which enables development on a modern basis. The initial set-up of such a project takes some time, however, by defining types in source code it allows the IDE to acknowledge errors at develop and compile-time instead of run-time. Therefore, it compensates with better readability and maintainability for complex projects. In addition to this, it enables IDE features like auto-completion of features of imported classes and a linter.

For the communication between server (back-end) and client (front-end), HTTP requests were chosen in a REST-like manner. The HTTP server was set up in Express[4], a minimalist web application framework for Node.js. In Express, HTTP utility methods, middlewares and endpoints were created. To validate the request data, express-validator[5] was used to build validation middlewares for every request. As a result, all preconditions for an endpoint were checked before sending the request to it. For a better maintainability, some logging was added using the well-tried logger library winston[6]. Since the prototype concentrates only on one issue management system and repository system provider first, GitHub was chosen as provider since it hosts both systems, and its developer API is described in detail. Therefore, the prototype's issue management system adapter and repository system adapter are using GitHub's Node.js REST client library Octokit[7] to connect with the GitHub API[8]. To test the application, the TypeScript version of the JavaScript testing framework Jest[9] was used. Although these libraries are the most important, others have been used, which are omitted here for clarity.

[2]https://nodejs.org/en/
[3]https://www.typescriptlang.org/
[4]https://expressjs.com/
[5]https://express-validator.github.io/docs/
[6]https://github.com/winstonjs/winston
[7]https://octokit.github.io/rest.js/
[8]https://developer.github.com/v3/
[9]https://jestjs.io/

The front-end prototype and mock-ups are programmed in Angular[10] with Angular Material Design[11]. Angular allows a developer to build cross platform single page progressive web applications and uses modern web tools to create some app-like experience. While Angular is a framework based on JavaScript, it is strong typed due to its use of TypeScript. In addition to this, the source code is well-structured by Angular, so a model-view-controller pattern can be applied. The HTML is separated from the Angular AppComponent code which binds the HTML view and model together. Model and view are synchronized automatically by Angular, so only the model must be changed. As a result, the front-end source code can remain maintainable even for a complex system.

Due to the short time available, the prototype could not be completed. To implement the Multi-Project Issue Modelling and Notation graph editor it is planned to use the Grapheditor Webcomponent[12] developed at the University of Stuttgart. This library dependency provides a standalone web component to realize rich and highly configurable graph editors. The edges and nodes are rendered with d3.js. Through CSS classes a dynamic styling is possible. The graph editor allows drag and drop as well as zooming and many custom events. In addition to this, the library allows text wrapping in plain SVG. Since the graph editor behaves like a standard HTML tag, the library can be used within all web frameworks.

[10]https://angular.io/
[11]https://material.angular.io/
[12]https://www.npmjs.com/package/@ustutt/grapheditor-webcomponent

Chapter 5

Validation

This chapter discusses the evaluation and validation of this thesis. First, in Section 5.1 a structured process to evaluate this thesis is depicted and described. Afterwards, a Goal-Question-Metric plan for evaluation is outlined. Section 5.3 reports the conducted expert survey. This section describes the procedure of the expert survey, how experts were identifies and addressed, and presents the artefacts provided for the experts. Furthermore, the feedback and results of the expert survey are stated out. Some experts provided valuable feature requests which are outlined in this section too. Subsequently, based on the evaluation the validation of this thesis is discussed. Finally, threats to validity are outlined in Section 5.5.

5.1 Overview of the Validation Process

The validation was carried out using a structured process, which is shown in Figure 5.1. As input for the validation process, a detailed description of the thesis' concept and system and the implementation mock-ups are provided. The input is described in Section 5.3.1. Two methods were used to obtain a qualitative validation. First a Goal-Question-Metric plan was elaborated, which contains 3 goals and built 3 specific validation questions. This GQM plan is explained in detail in Section 5.2. Afterwards, an expert survey, described in Section 5.3, was conducted. The experts were selected from large corporations as well as medium-sized companies and all work in areas where they must interact with several teams and projects. Using metrics, the process's output shows how helpful the system is, as well as possible opportunities for adaptation in industry.

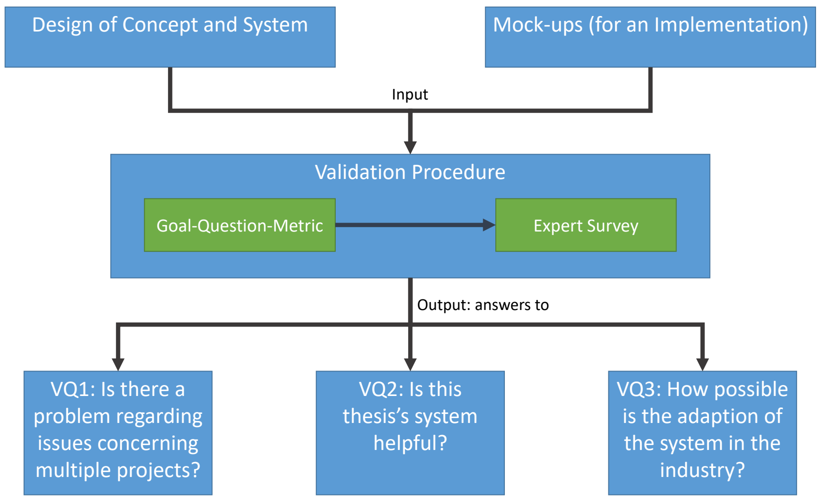

Design of Concept and System
Mock-ups (for an Implementation)
Input
Validation Procedure
Goal-Question-Metric
Expert Survey
Output: answers to
VQ1: Is there a problem regarding issues concerning multiple projects?
VQ2: Is this thesis's system helpful?
VQ3: How possible is the adaption of the system in the industry?

Figure 5.1: Structured process of the validation

5.2 Validation Goals, Questions Metrics (GQM)

The thesis' system's applicability and success comes down to the fulfilment of three major goals of (G1) flexibility in creating coding issues, concerning several projects, (G2) low effort managing coding issues, concerning several projects and (G3) extending coding issues with meta data, such as links to other coding issues and non-functional requirements. Flexibility and low effort could be achieved by using a graph editor to manage and create coding issues for one or more project components. The impacts of (G1) and (G2) can only be observed in comparison with other state of the art issue management systems. For (G3), the impact is hard to observe right now, since common issue management systems do not provide such meta data for their own. Therefore, issue links and non-functional requirements would have to be simulated in the normal coding issue text body. In order to completely address (G1) and (G2), it is necessary to evaluate if there is indeed a problem in microservice architectures when managing issues for several services. Subsequently, it must be examined whether the concept of this thesis correctly addresses and solves such a problem. Finally, it must be examined whether the industry can imagine using such a system productively to solve the problem.

This leads to the following validation questions:

(VQ1) Is there a problem in microservice architectures to adequately manage cross-service/project issues?

(VQ2) Does the thesis' concept of a multi-team, multi-project coding issue management system solve such a problem appropriate?

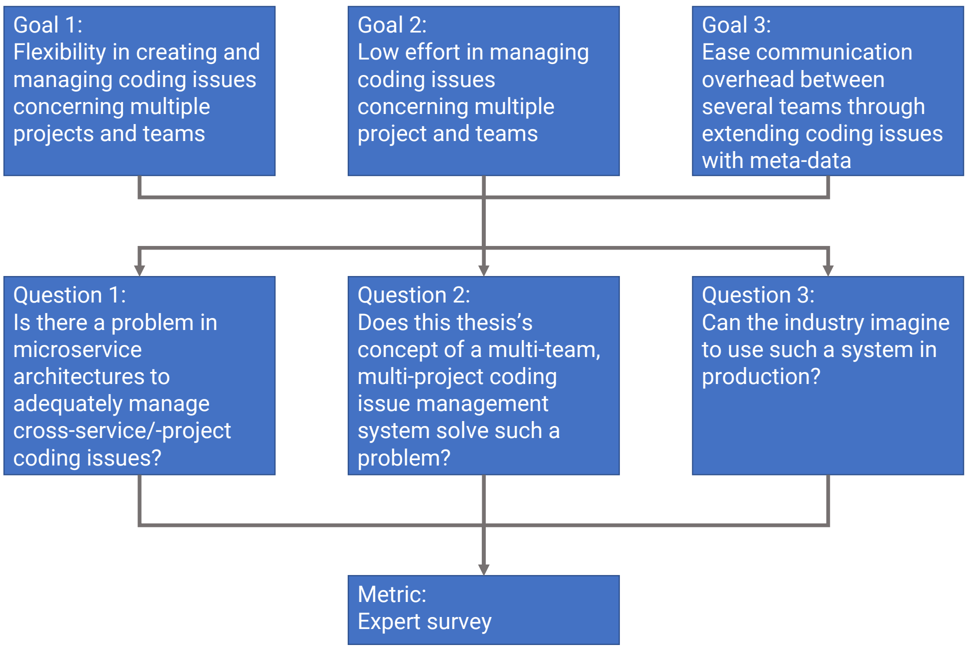

Goal 1:
Flexibility in creating and managing coding issues concerning multiple projects and teams
Goal 2:
Low effort in managing coding issues concerning multiple project and teams
Goal 3:
Ease communication overhead between several teams through extending coding issues with meta-data
Question 1:
Is there a problem in microservice architectures to adequately manage cross-service/-project coding issues?
Question 2:
Does this thesis's concept of a multi-team, multi-project coding issue management system solve such a problem?
Question 3:
Can the industry imagine to use such a system in production?
Metric:
Expert survey

Figure 5.2: Applied Goal-Question-Metric Plan

(VQ3) Can the industry imagine using such a system in production?

In order to address the three validation questions within the scope of this thesis, an expert survey was conducted as metric. The expert survey was chosen because other procedures would have been difficult or impossible to carry out at the time of the evaluation. For the expert survey, several software developers, cloud architects and project managers in the industry were conducted. The outcome of the expert survey is shown and discussed in Section 5.3. To summarize, a GQM plan was built from the point of view of a software developer, cloud architect or project manager. This plan is shown in Figure 5.2

5.3 Expert Survey

This section describes the expert survey conducted in order to evaluate and validate this thesis' concept and implementation. First, the survey procedure is outlined in Section 5.3.1. Afterwards, Section 5.3.2 delineates the results containing evaluation results and feature requests.

5.3.1 Procedure

This section describes, how the expert survey was conducted. First, the experts are characterized. Subsequently, this section describes how they were selected and which artefacts they received to evaluate this thesis' concept.

The experts are working as software developers, software architects or project manager. They often, but not exclusively work in a service-oriented problem area. Their working experience differ from 1 year after their studies up to more than 20 years. However, most of them work for more than 3 years after their studies in their jobs. Most of the experts were chosen, since they are industry contacts. First, industry contacts in this area of software engineering were asked, if they could evaluate the concept of this thesis, and if they know any other industry experts in this area who would be available for some evaluation. Sometimes, they could forward request to their colleagues. I have tried to cover both large software companies and small mid-size companies in order to get good coverage of potential users. Since not every large software company allows their workers to participate in such evaluation officially, some of the feedback was provided with a request for anonymity, so all experts are handled anonymously in this thesis. In total, feedback from Microsoft Corporation, Daimler AG, Motius GmbH, ibb testing GmbH, Novatec Consulting GmbH, and a big network and smartphone concern which has to stay anonymously due to legal reasons could be collected. In addition to that, the software developers writing their issues about having no multi-project system in Jira and Redmine forums were tried to contacted, as stated out in Section 2.2.3.

For gathering the expert feedback, a detailed e-mail with description of the concept and mock-up pictures of the tool (cf. Section 4.2) to support the description graphically was written. The e-mails were written in German or English, depending on the native or preferred language of the expert. Key question of the e-mail was whether the experts could imagine using such a system and if it solves a real problem. However, this process does not apply to the software developers of the Jira and Redmine forums, since there is no possibility to send them a message or get the e-mail addresses. Therefore, to contact them I responded on their forum threats in hope to retrieve an answer there. However, the Jira forums deleted the forum post after a few days.

First, the e-mail asked for some evaluation of the system in order to validate my work. The developers were told, that they could provide their feedback anonymously in case their employer would not approve it. Afterwards, a short description of the problem area the system can be used in, and use cases were provided. This enables the problem area of this thesis to be clarified at the beginning in order to avoid misunderstandings or ambiguities. Then, a detailed description followed, starting with a description of a multi-project coding issue and the overall functionality of the planned system. The mock-ups described the core features and clicks, so that the experts could create some look and feel experience. As core feature, the functionality of the graph editor, components, component interfaces are explained as well as issue management systems.

The next section of the description was about features and properties of multi-project issues. It explained the visibility and classification into feature requests, bug reports and unclassified. Furthermore, it was stated out, that issues could be classified automatically through NLP in future. Then the links to other issues and artefacts are described. In addition to this, non-functional requirements and possible features coming with them, e.g. automatic reject of pull requests or check of interface request latency, are explained. The end of this section, the notifications of coding issues were introduced.

In order to get a better imagination about the usage of system, both the right management and editing a coding issue are explained. Additionally, coding issue shadows, and how they work were outlined. Finally, the grouping and filtering of coding issues of the same kind in the graph editor are described.

As addition to the description of the system, the mock-ups showed some UI pages and dialogues. The mock-ups provided are shown in Section 4.2. There were 11 mock-up pictures which are described in the following:

- The first one, was a page containing the component view. The selected project contained 4 components, which were connected with each other, similar to the mocks showed in Chapter 4. The filter toggle buttons for feature requests and bug reports were activated, so the expert could see all feature requests and bug reports in the view. The mock-up showed both, bug reports and feature requests as public or private. Additional, shadows and resolves coding issues, if they were linked were shown. Some issues were grouped to show the grouping of coding issues of the same type.
- The second mock-up showed the dialogue view of a coding issue, which should appear, if a coding issue is selected in the component view. The dialogue shows the coding issue's details and allows the user to activate notifications, add assignees or labels, comment or close the coding issue. There is also a button to edit the coding issue.
- The third mock-up showed a dialogue, which appears, when a coding issue group is selected. The dialogue contains a list of all coding issues in this group. However only the title is shown.
- The next three mock-ups showed the dialogue to create a new component.
- Afterwards, the dialogue with component details is showed. A user can create new coding issues or a new interface for the component there.

- Mock-up 8 showed the detail dialogue of a component interface. Like a component detail dialogue, a user can create new coding issues for the interface here.
- Mock-up 9 showed the component view with activated notification toggle for feature requests. Therefore, only feature requests with notifications for the user are shown.
- The list view, where coding issues are shown in a list for each component, was shown in the 10th mock-up.
- The last mock-up showed the view to create a new coding issue.

The forum post for the software developer in the Jira and Redmine forums contained an equivalent description and the mock-up drafts. In case some industry expert had some questions, they were answered in detail, which sometimes led to a discussion of possible features. The feedback and feature requests of the experts was gathered and systematically ordered. Based on the ordered feedback and feature requests two mind maps were created, which can be seen in Figure 5.3 and Figure 5.4 of Section 5.3.2.

5.3.2 Results

In this section, the results of the industry experts' feedback are described. First, the feedback is explained in detail and the outcome is summarized in a mind map. Afterwards, the experts' feature requests are outlined and summarized in a second mind map.

Experts' Feedback

The overall expert feedback was in favour for the thesis' concept. The experts seem to like the basic idea, especially the graph editor, since it provides a nice overview of several projects and their coding issues. The experts mentioned an overhead of switching between several systems for each project using current systems. Some experts point out that the system could be particularly interesting for identifying blockers in the engineering process of a microservice architecture. They are adding that there is indeed currently a lack of a visualization for an entire microservice project where coding issues can be managed as well. One of the experts could also imagine an application for monoliths with strictly separated components.

The feedback of experts is shown in Figure 5.3 in a summarized overview as mind map and described in the following. The graph editor is the core feature of the concept implementation of this thesis. The experts acknowledged its possibility to give a nice overview of all projects/services where each project/service is shown as component. As already stated out, it could be useful to show and identify blockers. While it is a good “big picture”, especially for product managers, CTOs and lead architects, it may become somewhat confusing with large architectures. However, the experts think it should be suitable for up to 30 components. The visualization approach, like FMC diagrams or UML component diagrams, seems to be accepted and liked by the experts. They observed that it gives a quick qualitative overview of how many components and how many feature requests or bug reports exist in the entire system. The problem with the large architectures is not big in the opinion of some experts, since the filter possibilities can provide a good and clear overview. Keeping in mind roles as product managers, CTOs and lead architects, the experts acknowledge the system very useful for them. They add, that without such system, it is be hard to obtain not only a qualitative over all systems, but especially the development status of each service regarding the whole system. Another aspect of the graph editor is the strict separation of interfaces. The experts think it is a good way, since it is oriented to the MVC pattern, just for a microservice system view. This facilitates, in the opinion of the experts, assignments of stakeholders to be processed. The components can not only represent single services, moreover, they can map complex systems in the microservice architecture. The experts stated out that the component view itself follows a microservice way, since every team of a component can choose its own technologies and issue management system. In addition to this, they observe that the component view offers a nice way to create and manage coding issues concerning multiple projects. In the Experts’ opinion the multi project coding issues of this thesis’ concept offer a possibility to add monitoring systems. Regarding the concept of coding issue shadows, there are some concerns. The shadows seem to be to complex. One of the experts would rather pragmatically allow or reject the needed rights completely instead of creating coding issues. However, since the owner of a multi-project project most likely will not have the necessary rights himself to make this management changes, this solution does not seem to be feasible. Another expert stated out that there should only at most one shadow of a coding issue per multi-project project. Not only the number of parallel shadows seem to be a problem, the shadows and original coding issues should not also differ to much from each other. In summary, it can be said that for big microservice architectures the thesis’ concept can offer a great tool to manage the services and multi-project coding issues. The multi-project coding issues are in the Experts’ opinion the added value of the system by linking coding issues with architectures. Furthermore, through standardization no annoying switches of UIs are needed. Other standardizations, e.g. for coding issues seem to be helpful as well.

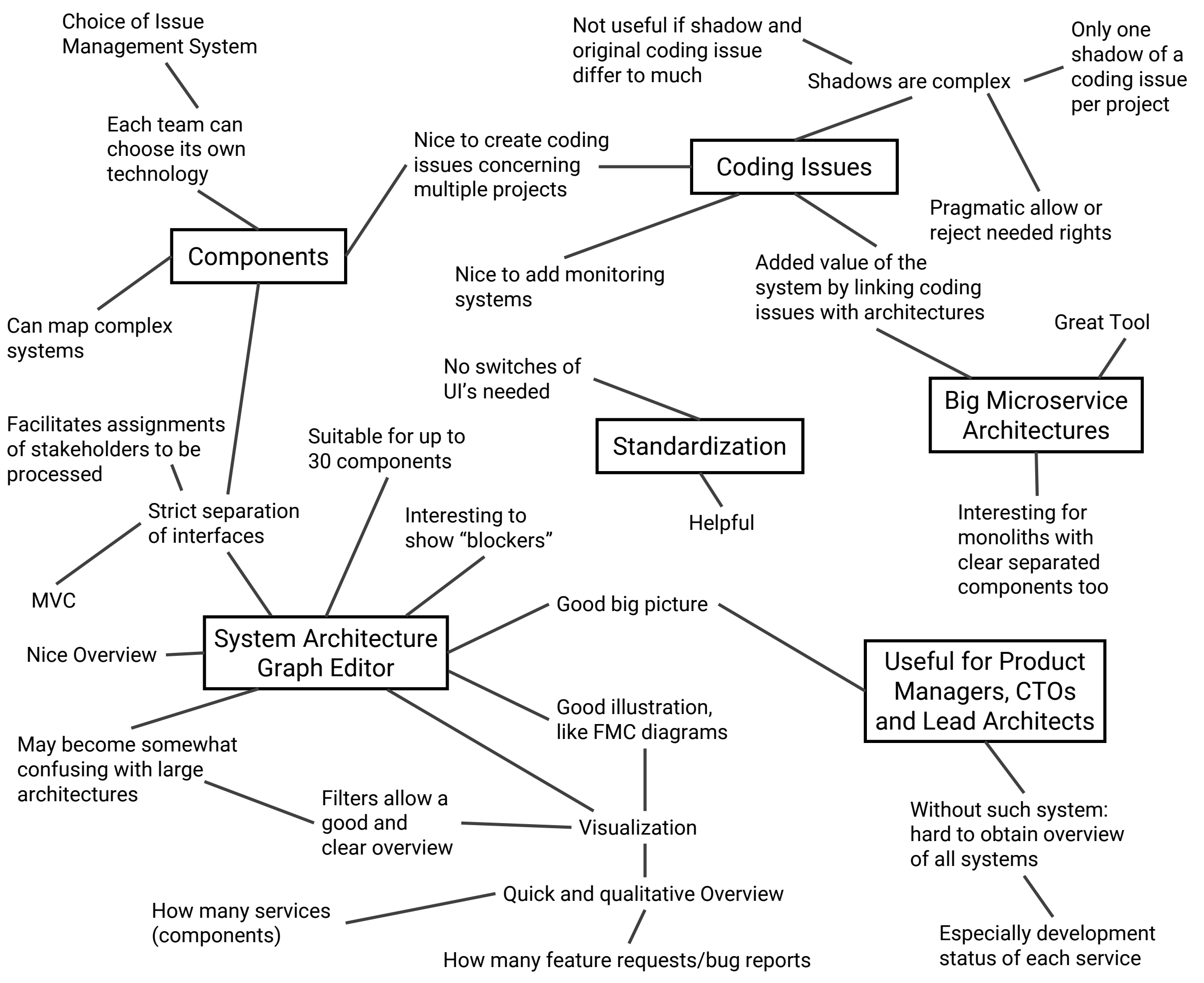

Choice of Issue Management System
Each team can choose its own technology
Components
Can map complex systems
Nice to create coding issues concerning multiple projects
Not useful if shadow and original coding issue differ to much
Shadows are complex
Only one shadow of a coding issue per project
Coding Issues
Pragmatic allow or reject needed rights
Nice to add monitoring systems
Added value of the system by linking coding issues with architectures
Great Tool
Big Microservice Architectures
No switches of UI's needed
Standardization
Helpful
Interesting for monoliths with clear separated components too
Facilitates assignments of stakeholders to be processed
Suitable for up to 30 components
Strict separation of interfaces
Interesting to show "blockers"
MVC
System Architecture Graph Editor
Nice Overview
Good big picture
Useful for Product Managers, CTOs and Lead Architects
Good illustration, like FMC diagrams
May become somewhat confusing with large architectures
Filters allow a good and clear overview
Visualization
Without such system: hard to obtain overview of all systems
Quick and qualitative Overview
How many services (components)
How many feature requests/bug reports
Especially development status of each service

Figure 5.3: Mind map of the Experts' feedback

Experts' Feature Requests

The Experts' not only gave some feedback for the concept and system idea, they also provided some feature requests they would like to have in such a system. These feature requests are shown in Figure 5.4 in a summarized overview as mind map and described in the following. For the components, the experts would like to have some hierarchical composition with zoom in or out. Such composition is already in the meta-model; however, a zoom functionality could be a great feature. In addition to this, some clustering of components as well as different views are desired. The views can make the graph editor more suitable for big microservice architectures with more than 30 services. The graphical composition of coding issues and components led some experts to request a few monitoring systems. In order to manage multi-project coding issues in an agile project, a multi-project Kanban/Scrum board was requested where developers of the multi-project project could see coding issues of multiple projects and subscribe to several components, so that they only see the coding issues of the components they are interested in. A product owner could manage the board for more than one teams, while each team can see only what its interested in. If a component is subscribed by multiple developers or teams, all of them can see the issues. There are also some standardization requests from the experts. They would appreciate standardized Tags and Labels for coding issues as well as coding issue names. Two experts mentioned an impact score and some prioritizing score could be a great add on for coding issues. Some distinction in low, medium, and high severity could help developers to concentrate on the most critical bug reports or feature requests. The impact score could be indicated through some colouring. There is also some colour indication requested for systems for which an above-average amount of coding issues have been opened recently (e.g. last 30 days) so problem zones can be seen fast. Some experts stated out that the introduction of personas or explicit roles to the system could be helpful, since there were only implicit roles at the moment of the evaluation. The colour indication and introduction of roles can lead to a fast identification of responsible persons in charge if coding issues have been increased significantly for a project. In summary all experts' feature requests are interesting ideas which can help to reduce communication overhead and manage multi-project coding issues. For the implementation, they all should be able to be included in the implementation of the thesis' concepts.

5.4 Validation

In this section, the validation of the thesis' concept and implementation is discussed on basis of the expert survey described in Section 5.3.1, and the results of the survey, which are described in Section 5.3.2. The thesis is validated with respect to the Goal-Question-Metric plan which was elaborated in Section 5.2. In regard to the validation questions of the plan, three hypothesises were created, one for each question:

(H1) If a software project contains several individual services/projects as in a microservice architecture, then there is a problem to manage issues appropriate.

(H2) If there is a problem and the thesis' concept is applied, then the management of issues in a microservice architecture is eased, so that this problem is solved.

(H3) When facing the problem of managing cross project and team coding issues, the industry could imagine to use this thesis' system.

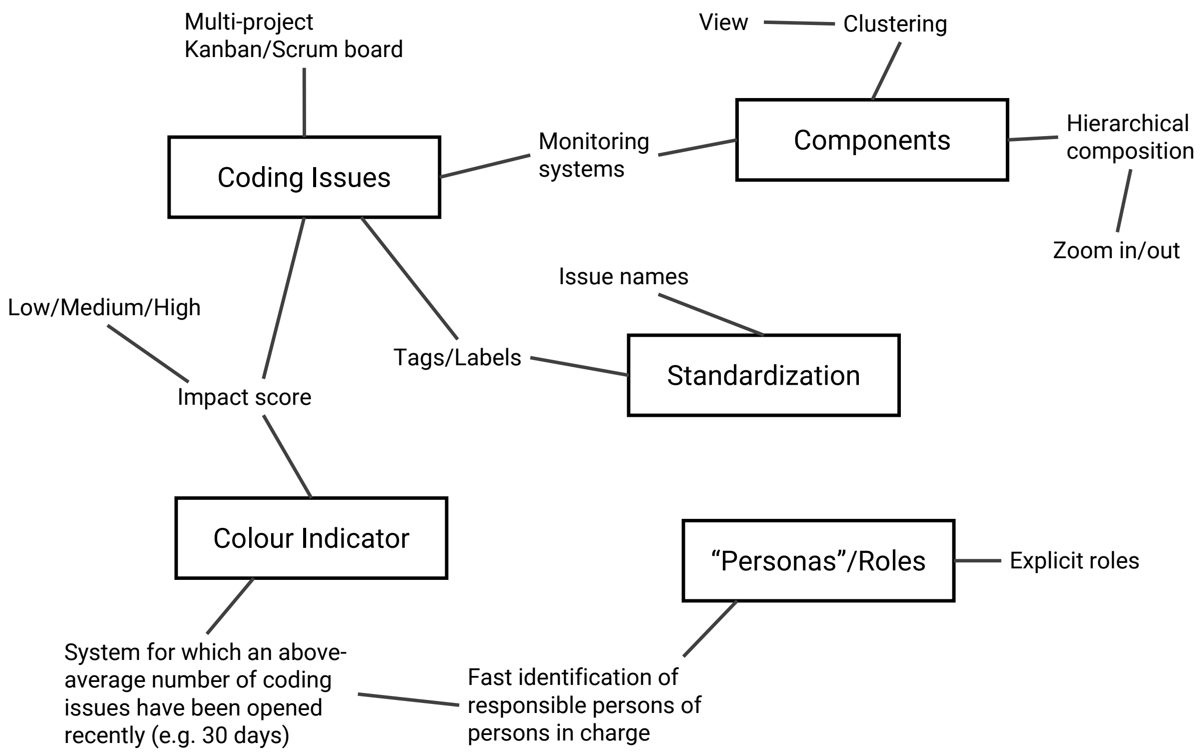


Figure 5.4: Mind map of the Experts' feature requests

Goal of this validation is to accept or reject the hypothesises to validate the thesis. However, it should be noted that the last two hypotheses are based on the first one. If the first hypothesis is rejected, the other two are no longer applicable.

The expert survey was conducted for in respect to a service-oriented architecture use case. The experts stated out, that managing issues in such an area is hard and comes with a huge communication overhead. The communication of issues concerning several services or teams are often done with meetings, instant messengers or via e-mails as described in Section 2.2.3. Therefore, those issues have to be created as coding issues in all concerning services' issue management systems. Clones and a technical debt is created in such case. Only when all services' coding issues are stored in Jira projects, multi-project coding issues can be simulated through K15t's backbone issues, which were presented in Section 2.2.3. In addition to the expert survey, several software developers are describing in Jira and Redmine forums a problem when it comes to managing issues for multiple projects or teams (cf. Section 2.2.3). Since all those points are in favour for (H1), there seems to be a real problem to manage multi-project issues appropriately. As a result, *hypothesis 1* is *accepted*: (H1).

Since (H1) has been accepted, (H2) can be applied. The thesis' concept introduces a possibility to create and manage multi-project coding issues. The thesis' system allows developers, and other stakeholders to manage them easily through a graph editor where each service is represented as component node. The results of the expert survey, shown in Section 5.3.2, state out, that the concept is applicable for big microservice architectures and eases the management of cross project and team coding issues. Since it is a common way to represent issues as coding issues, issues concerning more than one services can be represented qualitatively in thesis' concept and system. Therefore, the management of issues in a microservice architecture is eased, when using the thesis' concept, which solves the problem of (H1). As a result, the *hypothesis 2* is *accepted*: (H2).

Since (H1) and (H2) have been accepted, (H3) can be applied. As mentioned in Section 5.3.2, the Experts' feedback was in favour of using such a system. Some of the experts directly stated out, that they can not only imagine using the thesis' system a system, but would also like to use it in microservice architectures. Therefore, it seems the thesis' concept and system are accepted by the industry and that the industry could imagine to use it. As a result, the *hypothesis 3* is *accepted* : (H3).

All three hypotheses were accepted, which answered the validity questions of Section 5.2. However, further open questions arise regarding the use of the system:

- How much management effort can software architects save when using the thesis' system?
- Do software architects effectively benefit from creating issues for more than one project component?
- What are the limitations of the thesis' system?

These questions need additional surveys and can only be answered after a full implementation of the thesis' system.

5.5 Threats to Validity

This section describes the threats to validity of this thesis' concept. There are three types of threats. First, internal validity is discussed in Section 5.5.1. Afterwards, Section 5.5.2 reviews external validity. The construct validity is outlined in Section 5.5.3. Figure 5.5 depicts a summary of mentioned the threats to validity.

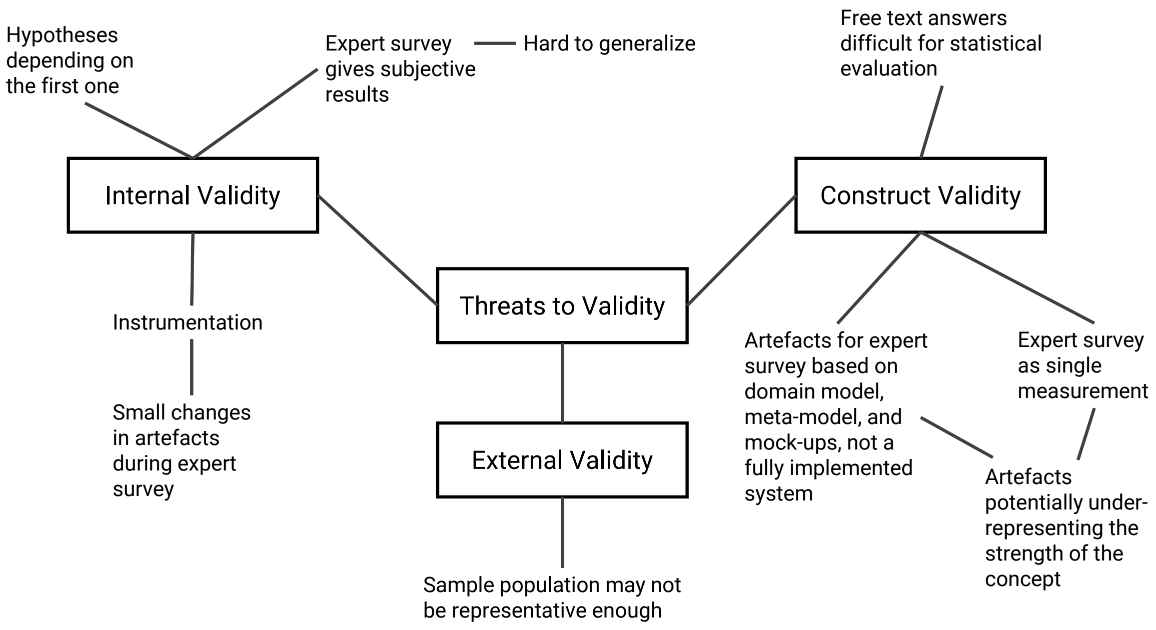

Hypotheses depending on the first one
Expert survey gives subjective results
Hard to generalize
Free text answers difficult for statistical evaluation
Internal Validity
Construct Validity
Threats to Validity
Instrumentation
Small changes in artefacts during expert survey
External Validity
Sample population may not be representative enough
Artefacts for expert survey based on domain model, meta-model, and mock-ups, not a fully implemented system
Expert survey as single measurement
Artefacts potentially under-representing the strength of the concept

Figure 5.5: Overview of threats to validity of this thesis' concept

5.5.1 Internal Validity

The evaluation conducted for this thesis comes with some internal validity threats because it is performed as expert survey. First of all, the three hypotheses are depending on the first one. For answering hypothesis (H1), some experts were asked before the actual expert survey, if there is a problem in managing issues for microservice architectures. Other experts stated out during the expert survey that there are some problems in managing issues for multiple teams or projects as in microservice architectures. However, since there are only a few answers for (H1), it is hard to generalize them to the complete population. Therefore, if they are wrong with their assumptions, (H2) and (H3) would no longer be applicable. Given that all experts come from several companies, are of different ages and work in different lengths, however, this risk should be minimal.

There is also some small instrumentation threat. The description of the system for later surveys was extended and detailed due to questions from the first experts. This could have led to changes in the outcome in comparison to the original result. However, it must be noted that the first experts have already received the same knowledge after the queries as the later experts. The results of the first experts were also comparable to those of the later experts.

Furthermore, an expert survey is always about subjective results. These results primarily reflect only the opinion of the respondents, which is subject to external influences. For this reason, special caution must be exercised when generalizing the results.

5.5.2 External Validity

There are also some threats to external validity regarding how representative the sample population is and, furthermore, the generalizability of the survey outcome. As experts, primarily former colleagues were asked to evaluate the thesis' concept and system, which could be biased. Although the experts come from both large corporations and medium-sized companies, the group of people surveyed is not completely representative. To cope with this threat, they were asked to forward the description and mock-ups for an evaluation to their colleagues so that a larger group of people could be created for the evaluation. Although the experts come from different big and small companies in Germany and Canada, only some feedback could be gathered, which could still open the possibility to discuss to what extent this group is representative enough. Since the sample group could not be generalized to the whole population, the generalization of the outcome of the validation, is also threatened. For a statistically more significant validation, another larger study should therefore be carried out. Within the scope and context of this thesis, however, the sample group of persons should be sufficiently representative enough.

5.5.3 Construct Validity

First of all, the description of the system are based on the domain, model, meta-model, and mock-ups and not on an implemented system. Therefore, the real system could differ a bit from the description. However, important components have not been clearly thought out and described well, so the description should be detailed enough. An implementation can follow with use of a certain amount of time on basis of the description and mock-ups are already implemented for the pictures in order to cope with this threat.

Another aspect is the quality of the evaluation itself. The evaluation is based on an expert survey. Therefore, just a single measurement was used. The artefacts for the expert survey potentially under-representing the strength of this thesis' concept to manage multi-project issues, which makes it possible, that the evaluation ignore the potential real effect. Additionally, the answers to the survey were given as free text, which makes statistical evaluation difficult. However, both the description of the system, the mock-ups, and the answers of the experts are very precise, which should prevent this threat.

Chapter 6

Conclusion and Future Work

This chapter summarizes in Section 6.1 the key aspects and insights of this thesis. Afterwards, Section 6.1 states out who benefits from the results an how. Finally, future work on basis of this thesis is discussed in Section 6.2. It includes which follow-up activities are suggested and why.

6.1 Results and Conclusions

This section summarizes key aspects and insights of this work. First, it briefly describes the problem statement which is solved by this thesis. Afterwards, the solution approach conceptualized in this thesis is presented in summary. Finally, an educated discussion showing the benefits of this concept is given which summarizes who benefits from this thesis' results and how.

Microservice architectures consists of several independent developed and operated services. Each of these services often is managed by its own project. These Services can dependent on other services by accessing their interfaces, e.g. a REST over HTTP interface. However, there are issues concerning multiple projects or teams if bugs, model changes or design decisions are relevant to more than one project. For example, if one service has a bug in its interface all dependent services' functionality might break. Current approaches for communicating such a cross-project issue exist. Developers or project owner communicate cross-team issues through e-mail, instant messengers, calls or meetings. As a result, a huge communication overhead arises. Therefore, current approaches cannot solve the problem statement in a qualitative way.

As a solution approach without communication overhead, this thesis suggests communicating cross-project or cross-team issues as coding issues. As main contribution multi-project coding issues are introduced. Multi-project coding issues are coding issues which can in contrast to normal coding issues concern more than one project/service. These multi-project coding issues are extended with meta-data. As first type of meta-data, they can directly link to other (multi-project) coding issues even if they are concerning another project/service. Instead of showing the link in the coding issue, the linked coding issue's data is depicted itself. This allows a developer to gain a fast-comprehensive overview of the issue to be solved. Furthermore, a multi-project coding issue can have traceability links to artefacts, such as source code snippets or model excepts, e.g. parts of an interface definition model. As last type of meta-data non-functional requirements can be added to multi-project coding issues as contracts to be fulfilled. These non-functional requirements can improve quality-of-service properties. They serve as restrictions that must be fulfilled, for example, so that a pull request can be accepted. In addition to multi-project coding issues, a domain model for a multi-project coding issue management in microservice architectures is outlined. Finally, a modelling language to notate multi-project coding issues and microservice architectures in a service architecture graph together is proposed. This modelling language is called Multi-Project Issue Management and Notation. To test this approach a multi-project coding issue management system prototype was implemented. However, within the timeframe of this thesis the implementation could not be finished. To counteract, system mock-ups were created to evaluate such a system. This thesis provides an overview of the implementation's architecture and main features. Additionally, the implemented parts of the prototype are described, and the mock-ups are depicted. In order to validate this thesis' concept, an evaluation of the concept and implementation was performed based on a Goal-Question-Metric plan and an expert survey.

This thesis' concept proposes a solution approach to manage multi-project and multi-team issues for microservice architectures in a qualitative way without huge communication overhead. The concept extends coding issues as a commonly used tool so that they can concern several projects/services. Therefore, development teams, project owner and lead architects of microservice architectures can benefit from this thesis' results. Project owner and lead architects can use such the Multi-Project Issue Management and Notation in order to maintain a qualitative overview of the overall project. They can create, manage, and monitor cross-service coding issues. Due to traceability links, issues can be better described. This allows developers to get all the information they need faster. The non-functional properties increase the product quality, from which not only product owners would benefit.

6.2 Future Research Challenges

This section outlines based on this thesis which follow-up research activities are suggest and why. It illustrates the challenges which must be solved.

First, a fully functional implementation of this thesis' concept must be built. This implementation should already contain the additional features suggested experts in an expert survey. Therefore, explicit roles and "personas" should be introduced to the system in order to allow a fast identification of responsible persons or persons in charge. Additionally, colour indication for (multi-project) coding issues for an impact score and other metrics can be helpful. Furthermore, the framework should have a multi-project Kanban/Scrum board to visually depict work at various stages of a process for multiple projects instead of one board for each project.

To benefit best from non-functional requirements, monitoring systems must be able to be plugged to the system in order to check these requirements. For these checks it would be a good feature to enable pull requests directly from the system. These pull requests can have the constraint, that all non-functional requirements must be fulfilled in order to be accepted.

Additionally, a two-phase commit for DevOps can be implemented. In case a developer team of a service executes a new commit, e.g. the team updates an interface, the developer teams of depending services should agree to the commit before carrying out the changes.

Right now, coding issues are only classified into bug reports and feature requests by hand. Classification of coding issues through NLP based on the coding issue's body text would be helpful. The coding issue's body text can be analysed for features that cannot be fulfilled, for example if a mentioned container version does not provide necessary features. In such case a warning can be shown for this coding issue. There is some other automation possible too. For example, automatic creation of coding issues for dependent services if an interface description model or interface source code of another providing service changes.

Furthermore, regression analysis is possible at least for service interfaces. The service interfaces' versions can be analysed for e.g. performance or bugs over time. Since multi-project coding issues are linked to the source code or models the analysis can determine which multi-project coding issues improved the quality-of-service and which worsened it. As a result, bad changes can be detected and improved.

The multi-project coding issue management system introduced in this thesis shows a service architecture graph. Deployment of these services can be realized to automatically test interface requests or show runtime logs for analysis.

Appendix

Bibliography

[BJS+08] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, T. Zimmermann. "What makes a good bug report?" In: *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*. ACM. 2008, pp. 308–318 (cit. on p. 8).

[CZZ+11] R. Capilla, O. Zimmermann, U. Zdun, P. Avgeriou, J. M. Küster. "An enhanced architectural knowledge metamodel linking architectural design decisions to other artifacts in the software engineering lifecycle." In: *European Conference on Software Architecture*. Springer. 2011, pp. 303–318 (cit. on pp. 15, 31).

[FH+01] M. Fowler, J. Highsmith, et al. "The agile manifesto." In: *Software Development* 9.8 (2001), pp. 28–35 (cit. on p. 11).

[FL15] M. Fowler, J. Lewis. "Microservices: Nur ein weiteres Konzept in der Softwarearchitektur oder mehr." In: *Objektspektrum* 1.2015 (2015), pp. 14–20 (cit. on p. 11).

[FLW16] S. W. Frank Leymann Christoph Fehling, J. Wettinger. "Native Cloud Applications: Why Virtual Machines, Images and Containers Miss the Point!" In: *Proceedings of the 6th International Conference on Cloud Computing and Service Science (CLOSER 2016)*. SciTePress, 2016, pp. 7–15 (cit. on p. 11).

[Fow17] M. Fowler. *Microservices Resource Guide*. 2017. URL: http://martinfowler.com/microservices (cit. on p. 11).

[HS14] M. Hammarberg, J. Sunden. *Kanban in action*. Manning Publications Co., 2014 (cit. on p. 9).

[HWH12] A. van Hoorn, J. Waller, W. Hasselbring. "Kieker: A Framework for Application Performance Monitoring and Dynamic Software Analysis." In: *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering (ICPE 2012)*. ACM, Apr. 2012, pp. 247–248 (cit. on p. 33).

[Iec11] I. Iec25010. “systems and software engineering–systems and software quality requirements and evaluation (square)–system and software quality models.” In: *International Organization for Standardization* 34 (2011), p. 2910 (cit. on pp. 22, 23).

[JPZ08] S. Just, R. Premraj, T. Zimmermann. “Towards the next generation of bug tracking systems.” In: *2008 IEEE Symposium on Visual Languages and Human-Centric Computing*. IEEE. 2008, pp. 82–85 (cit. on p. 8).

[KBB+09] B. Kitchenham, O. P. Brereton, D. Budgen, M. Turner, J. Bailey, S. Linkman. “Systematic literature reviews in software engineering–a systematic literature review.” In: *Information and software technology* 51.1 (2009), pp. 7–15 (cit. on p. 12).

[KPP+02] B. A. Kitchenham, S. L. Pfleeger, L. M. Pickard, P. W. Jones, D. C. Hoaglin, K. El Emam, J. Rosenberg. “Preliminary guidelines for empirical research in software engineering.” In: *IEEE Transactions on software engineering* 28.8 (2002), pp. 721–734 (cit. on p. 12).

[KZ10] P. Könemann, O. Zimmermann. “Linking design decisions to design models in model-based software development.” In: *European Conference on Software Architecture*. Springer. 2010, pp. 246–262 (cit. on pp. 14, 15).

[LL09] K. B. Laskey, K. Laskey. “Service oriented architecture.” In: *Wiley Interdisciplinary Reviews: Computational Statistics* 1.1 (2009), pp. 101–105 (cit. on p. 11).

[LL13] J. Ludewig, H. Lichter. *Software Engineering: Grundlagen, Menschen, Prozesse, Techniken*. dpunkt. verlag, 2013 (cit. on p. 8).

[LM12] J. Loeliger, M. McCullough. *Version Control with Git: Powerful tools and techniques for collaborative software development*. " O’Reilly Media, Inc.", 2012 (cit. on p. 10).

[MPH08] W. Maalej, D. Panagiotou, H.-J. Happel. “Towards Effective Management of Software Knowledge Exploiting the Semantic Wiki Paradigm.” In: *Software Engineering* 121 (2008), pp. 183–197 (cit. on p. 14).

[New15] S. Newman. *Building microservices*. "O’Reilly Media, Inc.", 2015 (cit. on p. 11).

[PCSF08] C. M. Pilato, B. Collins-Sussman, B. W. Fitzpatrick. *Version Control with Subversion: Next Generation Open Source Version Control*. " O’Reilly Media, Inc.", 2008 (cit. on p. 10).

[SS11] K. Schwaber, J. Sutherland. “The scrum guide.” In: *Scrum Alliance* 21 (2011), p. 19 (cit. on p. 1).

[Som11] I. Sommerville. *Software engineering*. Addison-Wesley/Pearson, 2011 (cit. on p. 8).

[Spe17] S. Speth. “Entwicklung von Microservices mit zusammensetzbaren API-Bausteinen.” B.S. thesis. 2017 (cit. on p. 11).

[Spi12] D. Spinellis. “Git.” In: *IEEE software* 29.3 (2012), pp. 100–101 (cit. on p. 10).

[ZWRH08] Y. Zhang, R. Witte, J. Rilling, V. Haarslev. “Ontological approach for the semantic recovery of traceability links between software artefacts.” In: *IET software* 2.3 (2008), pp. 185–203 (cit. on p. 13).

All links were last followed on November 24, 2019.

Declaration