Master's thesis at the Institut für Informatik der Freien Universität Berlin,

Software Engineering Research Group

Operationalizing the Architecture of an Agile Software Project

Arsenij E. Solovjev
Student number: 4223897
xeper000@gmail.com

Supervisor: Franz Zieris
Submitted to: Lutz Prechelt and Elfriede Fehr

Berlin, 08.12.2014

Abstract

Architectural erosion is a common problem among long lived software projects. One of the causes is a lack of understanding of the architecture of the given project. Also the agile idea of an "emergent architecture" is often misunderstood as an excuse not to take architecture into consideration when solving everyday problems. This problem can be addressed by operationalizing software architecture,i.e. the software architecture becomes itself an artifact of development and is integrated seamlessly into already existing agile practices. This thesis describes a practical approach to operationalize the architecture of the agile software development project Saros. A domain specific language for describing architecture called Archnemesis is introduced. A document written in Archnemesis is then used to perform architecture compliance checks whenever a developer submits code for review. At the time of writing this is the only open-source implementation of such a system to the knowledge of the author and can thus be used in different contexts as well.

Affidavit

I hereby swear an oath, that this thesis was done by none other than myself. All auxilliary material such as reports, books, websites and others are cited in the bibliography, quotes from other theses are marked as such. This thesis hasn't been presented to any other examining body and hasn't been published.

08.01.2014

Arsenij E. Solovjev

Acknowledgements

I would like to express deep gratitude to my thesis supervisor Franz Zieris, who has been a source of support, sound judgement and a guide through many difficult moments in this thesis.

This also goes for the head of our research group Prof. Dr. Lutz Prechelt, whose work as a professor, particularly his energetic lectures and commonsense approach sparkled my interest for software engineering.

Special thanks goes to the Saros team: Damla Durmaz, Stefan Rossbach, Arndt Lasarzik, Martin Wichner, Bernd Bieber, Maria Formisany and Holger Schmeisky. Without the feeling that we are all part of a bigger project, my dedication to this thesis would have been otherwise. Their support, active feedback and encouragement were vital for this thesis.

Last but not least, I would like to thank my parents without whose constant encouragement and financial support this would have been a much, much more difficult endeavor.

Contents

1 Introduction

But in practice master plans fail - because they create totalitarian order, not organic order. They are too rigid; they cannot easily adapt to the natural and unpredictable changes that inevitably arise in the life of a community.
Christopher Alexander, *The Oregon Experiment*

This introduction will begin by recounting the historical development of the concept of architecture in software and in agile software development in particular. After establishing what architecture is, I will address the problem of architectural erosion and the challenges of agile architecting which is the focus of this thesis. Following this will be a summary of previous work in this field, and after it differentiation of this thesis to the previous body of work along with the goal statement of this thesis. As is usual, a road map for the rest of this thesis follows, describing the different chapters ahead.

1.1 Brief history of Software Architecture

According to Wikipedia[1], architecture comes from the Greek *arkhitekton*, which literally means "chief builder". Ancient architecture as exemplified by Vitruvius based on three principles[2]:

- firmitas (stability, firmness)
- utilitas (utility, funcitonality)
- venustas (delight)

These can fairly easily be mapped to what is considered quality in software engineering. *Firmitas* pertains to reliability, security and maintainability, *utilitas* to the realm of requirements engineering and building the *right* system, whilst *venustas* is related to the realm of user experience. This is the parallel that James O. Coplien and Trygve Reenskaug present us in the introductory part of their article [JOC14], where they give interesting insights into the history of software architecture. So they write:

> [The architecture metaphor] originated with Fred Brooks in the 1960's. Brooks himself was a bit skeptical of hiw own brainchild [...]

[1] http://en.wikipedia.org/w/index.php?title=Architect&oldid=636293185
[2] `http://en.wikipedia.org/wiki/Vitruvius#Vitruvius.27_De_Architectura_libri_decem_.28De_Architectura.29`

> Software has strongly embraced this metaphor, both for its casual parallels to programming-in-the-large on one hand and for some of its specific techniques on the other.

The term was introduced to the world of software engineering by Fred Brooks as a mere metaphor (a few years apart from the engineering metaphor itself) the large-scale view of the organization of a software system. However, since then the concept of software architecture has taken a life of it's own, becoming a research field of it's own and "software architect" becoming a job description. Today many definitions for software architecture exist, however I will use the one most widely used in literature, provided by Garlan and Shaw [**?**, Garlan1993]

> [Software architecture is a level of design that] goes beyond the algorithms and data structures of the computation: designing and specifying the overall system structure emerges as a new kind of problem. Structural issues include gross organization and global control structure; protocols for communication, synchronization, and data access; assignment of functionality to design elements; physical distribution; composition of design elements; scaling and performance; and selection among design alternatives.

As you see software architecture in this context is a rather heavy-weight and all-encompassing concept. Modelling software architecture has been used to predict possible problems in an architecture, document important architectural decisions, developing architecture-based development methodologies and formalizing architectural styles using *architecture description languages* [**?**, Garlan1993] The point being that all of these approaches to architecture are primarily done top-down, they produce heavy documents, the methodologies take considerable time and expertise.

However, also another way to approach architecture in software is noted by James O. Coplien and Trygve Reenskaug. They write:

> The patterns discipline is an example of the latter ["and for some of it's specific techniques on the other" from the quote above], whose philosophies of local adaptation and piecemeal growth became an alternative to big-up-front-design in the 1990's and flourished in the guise of the agile movement in the ensuing decade.

The patterns discipline borrows it's philosophy from the work of the influential architect Christopher Alexander[3], most notably his book "A Pattern Language". In it Alexander introduces the notion of patterns:

[3]you will encounter the file extension "chralx" later on in this thesis, this is a tribute to Christopher Alexander

> Each pattern describes a problem that occurs over and over again in our environment, and then describes the core solution to that problem, in such a way that you can use the solution a million times over, without ever doing it the same way twice.

So each pattern (or design pattern) is an abstract solution to a class of problems, i.e. a configuration of form and function which resolves misfits in a whole class of problems. A pattern can be translated into a concrete solution for a concrete problem of that class of problems. A pattern language is thus a web of relationships between such patterns. By combining such patterns one can form complex solutions and *communicate* them more easily given the receiver is well-versed in design patterns. This was roughly the vision embraced by the patterns discipline in software engineering, spearheaded by the Gang of Four[GHJV94].

A paradigm shift occcured in software engineering with the advent of agile processes, as pioneered by the Agile Manifesto [BBvB$^+$] (chronologically around the same time as the patterns community came into the limelight). The Agile Manifesto is fairly short, so it's appropriate to include the whole text here (original emphasis has been preserved):

> We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:
>
> INDIVIDUALS AND INTERACTIONS over processes and tools
> WORKING SOFTWARE over comprehensive documentation
> CUSTOMER COLLABORATION over contract negotiation
> RESPONDING TO CHANGE over following a plan
>
> That is, while there is value in the items on the right, we value the items on the left more.

So, in the agile paradigm plans with big-up-front-designs are seen as confining and likely wrong in the long term, since agile practitioners expect that requirements will change, which is likely to render such an upfront design useless (or maybe even more importantly, make development more expensive). Instead of the meticulous planning up-front, typical of the architectural practices before the agile movement, agile practitioners focus in responding to change and collaborating with customers closely. Typically agile projects release often, these releases are then presented to the customer, feedback and consensus is achieved with the customer on what the next developments should be. As you see agile development is a flexible step-by-step process, kept on the right track by close customer collaboration. It is this required flexibility and small iterative steps that make agile processeses uninhabitable for the heavy-handed architectural practices which involve big upfront designs and large-scale semi-formal assessment techniques. This does not mean that agile development foregoes architecture completely, rather it deals with it in it's own way.

One of the original authors of the Agile Manifesto, Martin Fowler, describes a (non-)approach to architecture[Fow] he has often observed. In it design is emergent and architecture evolutionary. Design and architecture of a software system evolve with changing requirements and refactorings, as the need arises. Although this can be seen as a complete lack of design, Fowler argues that this depends on how much designing is actually done during development. Fowler does stress that design has to happen upfront in some portion, and that it has to anticipate future change and that coders have to be apt designers. How much of Fowler's advice is followed in the everday agile practice is an open question.

1.2 Problem statement

So, what happens if architecure is neglected? One scenario that Fowler describes is the "code and fix" process pattern, where development becomes a very tedious affair, due to constant bugs being introduced into the code, as the code does not have a clear structure. From an architectural standpoint the term for this is "architecture erosion", Terra et al.[**?**, Terra2012]escribe architecture erosion thusly:

> Software architecture erosion designates the progressive gap normally observed between the planned and the actual architecture of a software system as implemented by its source code. Although the causes for this architectural gap are diverse, ranging from conflicting requirements to deadline pressures, the consequences always include degradation in the internal quality of the system, with a negative impact on properties like maintainability, evolvability, extensibility, and reusability. When the process is accumulated over years, architectural erosion can transform software architectures into unmanageable monoliths.

Architectural erosion is a general problem with long-lived software projects. As the architecture decays the system becomes hard to understand and hard to maintain. The reason might be twofold, either the intended architecture is not adequate, or the developers, for whatever reason, have not followed the implementation of the architecture thoroughly. On either account architecture erosion poses a problem, especially in agile projects as the agile agenda gives no specific directions here (apart from Martin Fowlers wise advice).

1.3 Related Work

So, how has this problem been addressed before? How have agile methods and monolithic architectural practices been reconciled?

A series of recent research papers on the topic highlight the need of uniting the traditional approach to architecture [MABM13]. Here I present the ones I found most interesting for this thesis:

Stal[Sta14] further develops the ideas of Fowler on how to handle emergent architectures to a more immediate form and proposes a method for architectural refactoring as a means to prevent architectural erosion. He proposes that architectural refacotrings be done by a software architect, who can identify architectural problems, solves and evaluates these as the process goes on. He proposes that architectural refactorings be done simultaneously with other refactoring activity for the best value-add.

Cleland-Huang, Czauderna and Mirakhorli[CHCM14] suggest a light-weight elicitiation technique for architecturally significant requirements. Their approach is based on creating architecturally savvy personas[4]. They describe how these personas can be used for the elicitation of requirements as well as for the evaluation of architectural solutions.

Van der Ven and Bosch[vdVB14] make an empirical study on five projects in which they measure how the projects perform on a scale which measures who does the architecting (is there an architect or is this done by the development team), how long the feedback loop is between the making of an architectural decision and it's implementation and how architectural knowledge is shared (whether they are verbally communicated, held in meeting notes or use more formal template based documentation tools). They have found out that if architectural decisions are managed by the (coached) development team, are communicated directly and have a short feedback loop, this increases the chances of the project's success.

A thesis which has directly influenced this one is that of Anne Augustin [Aug13]. In it she develops an architecture description DSL and a checking mechanism which is integrated into the development process. It is upon seeing her presentation of her thesis that idea for this thesis was born.

[4]A fictional character created to represent a stakeholder in architectural decisions

Another thesis which is important concerns itself with architectural problems of Saros, the project which is the object of this thesis, and which I will introduce in more detail further in this chapter. In it Patrick Schlott[**?**] provides thorough analysis of Saros' architecture based on Kruchten's "4+1" view model on architecture [Kru95].

Another thesis whose object is Saros' architecture was written by Belousow[Bel11]. He does not implement any one solution, but develops a thorough road map no how to make Saros more modular by applying OSGi technology[5].

1.4 Goal statement

This thesis is much less general than the papers above, as it concerns itself with the architecture of one single project, Saros.

Saros is an Eclipse[6]-plugin for distributed pair programming. It is an open-source project developed in a research context at my alma mater FU Berlin. This means that most of the code is written by students who write their theses about Saros, as well as some diehard open-source developers. An inside joke says that Saros employs a new software methodology called thesis-driven development. As a consequence it has a very high fluctuation rate, i.e. a student develops his feature, defends his thesis and does not contribute to the project any longer (of course there are honorable exceptions to this rule). The project uses many agile processes, such as continuous integration and unit testing. It is also used by the university for curricular software projects, where students learn how to use a wider array of agile methodologies and practices, such as Scrum[7] and Kanban[8].

As a recent master's thesis[Sch13] has shown, Saros is subject to architecture erosion. The specific causes of this haven't been investigated, however I would argue this is an issue of the development process. New developments on Saros are being rigorously reviewed, and sometimes these address technical debt. However, technical debt, of which architectural erosion is a part, is not always obvious when it is commited to code, there is no comprehensive overview of technical debt and it is not being addressed systematically.

[5]OSGi is a modular platform, in which components of an application are very loosely coupled. Source: `http://en.wikipedia.org/w/index.php?title=OSGi&oldid=635766392`

[6]A well-established IDE

[7] An incremental agile software development framework `http://en.wikipedia.org/w/index.php?title=Scrum_%28software_development%29&oldid=636319630`

[8] A method for managing knowledge work with an emphasis on just-in-time delivery `http://en.wikipedia.org/w/index.php?title=Kanban_%28development%29&oldid=636357072`

In this I would like to improve the way architectural erosion is handled in the development process by applying the a well-established open-source static analysis tool (Sonarqube[9] on Saros. This would contribute to preventing architectural erosion, as well as support architectural changes in Saros. Using SonarQube it is possible to alert the developers of new architectural violations, to find and to inspect pre-existing violations.

Saros's build-infrastructure would be extended to convey differential architecture-checks upon a commit. This means the source code will be checked for conformance to certain architecture rules. This would be done by integrating SonarQube's architecture rule constraints into the development/build process. For the developer this would result in an additional criteria required to pass a gerrit review. This way a developer would need his patchset to be

1. compilable,
2. not failing any unit tests, and
3. comply to the architecture.

It would also be of interest to see how we could use some of the other features of Sonar, some of them overlap with existing build steps in the current build configuration. Currently Jenkins is responsible for a number of tasks such as computing code coverage, PMD source code analysis and FindBugs. All of these tasks belong to quality assurance, a field in which SonarQube excels, so it will be considered to delegate these tasks to SonarQube. This would leave Jenkins the responsibility to build, communicate with Gerrit, and to delegate to Sonarqube.

As of now the Saros architecture is documented on its homepage[10], in some theses as well as in the heads of it's developers. An aspiring Saros-developer mightnot be aware of these resources or could simply forget them. After my extension though, he would be forced to see whether his patchset makes sense from an architectural point of view and be forced to act upon that by either changing his patchsetor the architecture itself. The exact behaviour of the developer is unknown, and I would like to evaluate how this change would effect the current development process. Also it is to be evaluated if the effort needed to support this addition to the development process is worthwhile.
There are some obstacles to modifying the infrastructure in the desired manner, which I seek to overcome.

- Saros's architecture is not formulated in a series of formal rules/specification, but rather verbally and graphically. To overcome this, I would

[9] http://www.sonarqube.org/
[10] http://www.saros-project.org/architectureDocumentation

consult the existing documentation, some developers, as well as research Saros's commit message history on architectural ideas in Saros. With that knowledge I would proceed to create a basic set of rules, which should be extended and modified by future developers as is seen fit.

- SonarQube's capabilites in terms of defining architecture rules are limited to Sonar's interface. This poses an inconvenience, since if a developer would like to make changes to the architecture rules, he would be forced to use yet another tool in an already difficult process. A more preferable approach would be to have a DSL describing the architecture in the source code itself. This was already done in Anne Augustine's thesis[**?**, **?**, Augustin2013] however her approach is tailored to a specific build infrastructure and cannot be applied to Saros. Hence I would like to investigate the possibilities of extending Sonar to provide this feature (i.e. to be able to make architecture checks on the basis of a specific document within the source code).

- Sonar's capabilities in respect to architecture checking are limited to formulating dependency rules (i.e. allowing/denying access to specific packages and classes). Software architecture has more facets though, as for example in Kruchten's 4+1 View Model [Kru95] or architecture description languages. I would like to investigate the feasibility of implementing additional levels of checking.

1.5 Roadmap

Chapter 2 discusses architectural description languages, what they are and how they could be used for the modeling of Saros. Chapter 3 discusses Saros' infrastructure and the adjustments made for it to be able to support archiecture compliance checks. Chapter 4 tells the story of the implementation of the architecture compliance checks. Chapter 5 summarizes the work done by this thesis, discusses its strengths, weakness and possibilites for development proceeding from this thesis.

2 Architecture Description Languages

In the previous chapter a brief history of software architecture was introduced, along with the problem of architecture erosion and the goal statement of this thesis. In the goal statement I mention that I wish to explore in what other ways architecture can be described other than through simple layer checks. Luckily, such a technology exists and is called *architecture description language*(ADL). An ADL is a formal modeling notation that provides architectural specification, loosely defined "an ADL for software applications focuses on the high-level structure of the overall application rather than the implementation details of any specific source module"[MT00][Ves93]. They pose an interest for this thesis as they are expert achievements in modeling architecture. The hope behind studying ADLs for this thesis was to see whether one of them could be applied to describe Saros' architecture, and if not just to get a general idea of how software architecture is modeled.

In this chapter I will provide a brief overview of my study of ADLs. Apart from overviews such as [MT00] and [All97], which make comparisons of the most known ADLs, as well as describe their key features. In the following I will describe what is an ADL, how they differ, as well as provide (hopefully) some more in-depth insight into the three ADLs I mentioned.

In the concluding part of this chapter I would share my thoughts on how much of the ideas used in ADLs are feasibly applicable to the architecture of Saros.

It is of note, that in her master's thesis [Aug13] abandoned ADLs altogether, since the specification of interfaces was in conflict of the agile methods used by the company she developed her modelling language for. This goes in line with what Georg Buchgehe and Rainer Weinreich say about ADLs in their article "Continuous Software Architecture Analysis"[BW14].

> The creation of ADL-based architecture models is sometimes difficult and requires technical stakeholders with specific expertise. This may be one reason why ADLs have not yet found their way into mainstream software development. Additional reasons are listed by Woods and Hilliard and include the restrictive nature of ADLs, the lack of multiple views, lack of good tool support, their generic nature, and the lack of domain concepts

In fact in a paragraph below they suggest DSLs as the more common practice:

> In addition to ADLs, DSLs can be used to describe software architectures. Architecture-centric DSLs are typically developed for a particular domain or even a particular system [...]

I would not dismiss ADLs for now, as a system's architecture does not change significantly once a certain level of maturity has been achieved, therefore it is of no substantial hindrance to have a "heavyweight" description of an architecture, assuming this description can be modified incrementally, as need arises.

2.1 Criteria

Which brings me to the open question of what I consider to be useful in an ADL useful. Ideally, a formalized architectural description would be

- precise enough to capture essential architectural constraints and structure,
- general enough, so that it doesn't have to be modified too often,
- understandable enough to be maintainable and modifiable, and
- simple enough for a compliance check to be implementable without greater effort.

2.2 Overview of different ADLs

A comprehensive overview of the features of ADLs can be found in the article of Nenad Medvidovic and Richard N. Taylor[MT00]. In it they classify and compare different ADLs, and in the introductory part of the article they provide a comprehensive overview of the typical elements that ADLs have:

Component is a unit of computation or data, it can be as small as a single procedure or an entire application. A component can have:

- **Interface** is the set of interaction points between a component and the external world. It can be used to specify the services a component needs and provides.
- **Types** are an attribute of components. Most ADLs allow to define an abstract type, which then can be instantiated multiple times in the ADL
- **Semantics** enable analysis, constraint enforcement and mappings of architectures
- **Constraints** force certain assertions or properties on components. Though most ADLs have this implicitly through the use of interfaces, some goe beyond that and have a separate constraint language.

Connectors are architectural building blocks that model interactions among components. They also possess the same features as components:

Interfaces define a set of interaction points between the connector and the components it connects.

Types are also used for the reusability of connectors. As connectors are often complex protocols it makes sense to have them in extensible type systems.

Semantics for protocols and transactions are also typically modeled, in otder to enable constraints on them.

Constraints are used in order to ensure adherence to protocols, set usage boundaries and establish intra-connector dependencies.

Typcally an ADL subsumes a formal semantic theory, such as Petri nets, Statecharts, partially-ordered event sets, CSP etc.

Let's take a closer look at Petri Nets. According to Wikipedia[11] a Petri net is "one of several mathematical modeling languages for the description of distributed systems. A Petri net is a directed bipartite graph, in which the nodes represent transitions (i.e. events that may occur, signified by bars) and places (i.e. conditions, signified by circles). The directed arcs describe which places are pre- and/or postconditions for which transitions (signified by arrows)." In the context of architecture nodes can be viewed as components maintaining state, transitions as operations of components and vertices as simple connectors. Of course this has limitations as Petri Nets cannot specify interfaces, and every state component has to be a processing component. However, other formalisms such as partially-ordered sevent sets, CSP, Obj and Anna have been successfully used in ADLs[MT00].

2.3 Applicability to Saros and Further Reading

In conclusion however, the pursuit to find a suitable ADL for Saros was not succesful. ADLs are heavyweight, the tools used are not freely available and it seems to require a great level of expertise to extract an architectural description for Saros, as Saros is already implemented. The other way around, first writing an architectural description and then implementing Saros might have been easier. However rewriting Saros for the sake of having an architectural description of it does not seem like a cost-effective idea.

For this thesis I decided to go along the lines of what Buchgeher and Weinreich were saying, and instead use a DSL to describe Saros' architecture. However some insights on how ADLs work and what elements they use have proved to be useful concepts to have in one's head when trying to create a DSL. This is what I learned from studying ADLs:

[11] http://en.wikipedia.org/wiki/Petri_net

- Almost all ADLs use these three basic elements: *component*, *connector* and *constraint*.
- While describing components and connectors is really close to a simple layer definition, constraints are the concept that goes beyond them and as such is interesting to me.
- Unfortunately, the description of constraints in ADLs usually involves some degree of formalism, which seems convoluted beyond some point, as understanding it seems as hard as understanding Saros' architecture without an architectural description.
- However a simplified version of such constraints is something to look out for.

3 Integration into the development process

If I have seen further than others, it is by standing upon the shoulders of giants.

Isaac Newton

In the previous chapter the general problem of architecture erosion and in architecture compliance checking as a solution was proposed. In order to implement architecture compliance checking it was necessary to lay the groundwork in the infrastructure and the development process of the project. The infrastructure of Saros is based on three systems which work in concert

- Git for version control
- Jenkins for continuous integration and
- Gerrit for peer reviews

In order to understand where architectural compliance checking fits within the development process, one has to understand these systems and their interplay.

This chapter will introduce the Continuous Integration system Jenkins as well as the review system Gerrit. Further it will be discussed how the systems are integrated and how they are used in development. A fourth system called Sonarqube is introduced into this infrastructure. The chapter finishes off with a more in-depth description on how integration between Jenkins, Gerrit and Sonarqube was achieved. This integration is one of the more work-intense parts of this thesis.

3.1 Saros' infrastructure

3.1.1 Continuous Integration with Jenkins

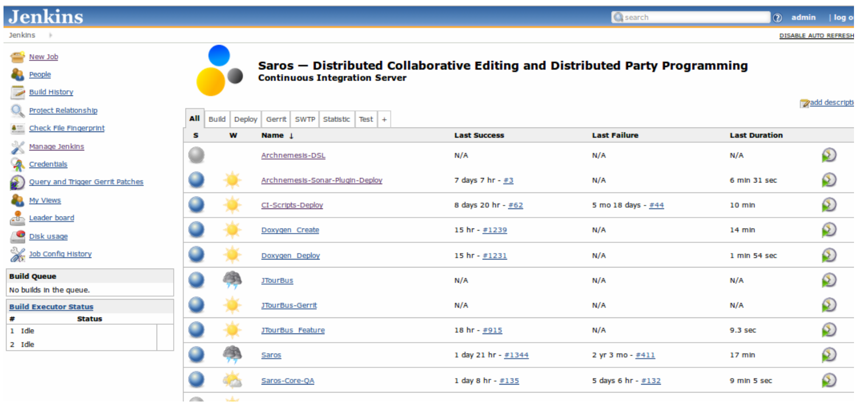
Continuous Integration (CI) is an agile practice first proposed by Grady Booch and brought to it's more wide-spread form within the methodology of eXtreme Programming [12]. In the agile context, continuous integration allows developers a more finegrained quality control over the developed software, especially when more than one developer works on the project. This last aspect highlights the problem that CI addresses, when more than developer works on the code, each developer usually works on a possibly different version of the code and also produces a unique version of the code. Before CI the practice was to merge all development efforts into one code base, and release it. This kind of practice has since then been called a "Big Bang

[12]http://en.wikipedia.org/w/index.php?title=Continuous$_i$*ntegration&oldid* = 636359088

Merge", as it can lead to many integration conflicts, [13] which is more often than not problematic.

As a remedy to reduce risk in such situation CI prescribes than changes to the code are commited often and that these are automatically integrated into a single source code repository. That way any integration problems are spotted as soon as they are commited to the code base.

Jenkins is an open-source system for CI. It runs as a server-side application, typically on a dedicated machine accessible to everyone in the development team. The most important concept used in Jenkins, is the concept of a job. A job is a task with a uniform configuration that Jenkins executes, a single execution of a job is colloquially called a build[14]. A simple example for a job would be when Jenkins polls for updates on the master branch (i.e. the branch that contains the code that will be released), if an update is present Jenkins runs the build scripts for the updated master branch. If the build was successful the build is marked as successful, if not it is marked as a failure. The team then knows that the latest patch has "broken the build", and is in position to amend the patch in due time. Typically, job configurations are much more versatile than the one described above. Plugins can run test suites or static analysis tools, these in turn can influence the status of a build, thus providing feedback on the quality of the last commit.



Saros — Distributed Collaborative Editing and Distributed Party Programming
Continuous Integration Server

Build Queue
No builds in the queue.

Build Executor Status

#	Status
1	Idle
2	Idle

S	W	Name ↓	Last Success	Last Failure	Last Duration
●		Archnemesis-DSL	N/A	N/A	N/A
●	☀	Archnemesis-Sonar-Plugin-Deploy	7 days 7 hr - #3	N/A	6 min 31 sec
●	☀	CI-Scripts-Deploy	8 days 20 hr - #62	5 mo 18 days - #44	10 min
●	☀	Doxygen_Create	15 hr - #1239	N/A	14 min
●	☀	Doxygen_Deploy	15 hr - #1231	N/A	1 min 54 sec
●	⛈	JTourBus	N/A	N/A	N/A
●	☀	JTourBus-Gerrit	N/A	N/A	N/A
●	☀	JTourBus_Feature	18 hr - #915	N/A	9.3 sec
●	⛈	Saros	1 day 21 hr - #1344	2 yr 3 mo - #411	17 min
●	⛅	Saros-Core-QA	1 day 8 hr - #135	5 days 6 hr - #132	9 min 5 sec

Figure 1: Jenkins' web interface. Below the project name you can see a list of jobs, with the spheres indicating the status of the last build (blue for success, yellow for unstable, red for failure, grey for disabled)

[13]Simply put, an integration conflict occurs when two branches are merged together and have changed the (previously) same behaviour and/or structure of the system. Thus there is a fair chance that the new version of the system will not behave as intended.

[14]Technically, a build is the generation of executables from source code. Not all Jenkins jobs generate executables.

The second important concept in Jenkins to know for our purposes, is the concept of a workspace. Each build has a workspace which contains the complete code for the project it is. This is where all build, test and other scripts are run.

3.1.2 Peer reviews with Gerrit

Peer reviews are a good practice for both quality assurance and knowledge distribution in teams. Basically, a patch is put up for review from other team members. Only after it has been approved is it accepted into a master branch. Usually the code undergoes a series of revisions until it is finally accepted. A review can have a varying degree of formalism involved. Through reviews bugs and design flaws can be detected early, also by looking through the code the other team members gain knowledge about the code.

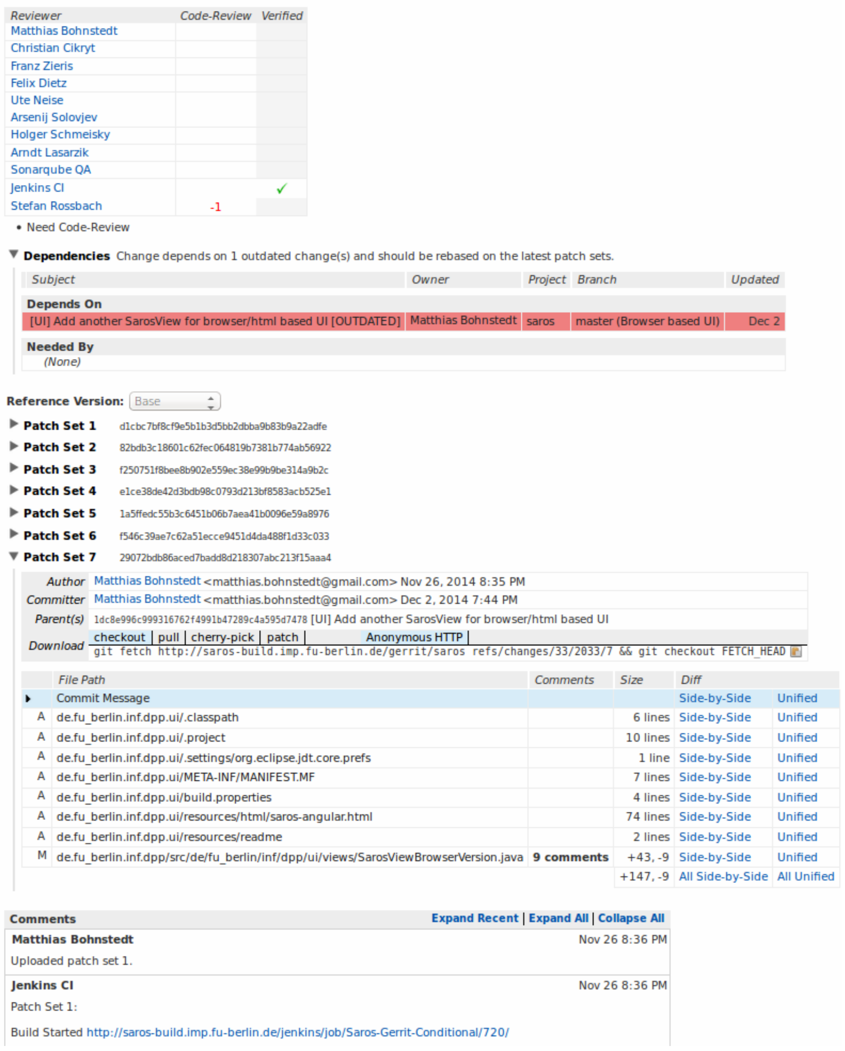
Gerrit is a Git-based[15] peer review board. Developers work on their patches locally and then publish them to Gerrit where they are available for review. Another developer reviews and scores the patch. The diffs of the patch are visible through the web interface and peers can comment on single lines of code. After a patch is approved by peers it can be merged into the master branch. Approval is managed by giving patches a positive or negative score in a given score category. You can see how this looks in Gerrit's web interface in figure 3.

3.1.3 Development process and interaction within the infrastructure

In the context of Saros' infrastructure, Jenkins and Gerrit are integrated in the following manner: Gerrit is able to notify Jenkins that a new patch has been pushed for review. In turn Jenkins starts a build of the patch, after the build scripts have run, unit tests are executed. Depending on the success of the build, Jenkins posts a review to Gerrit giving a score of +1 or -1 for the "Verified" label. Here is a quick outline of the complete process from a developers point of view (shown as a diagram in figure 4):

1. Developer pushes patch to Gerrit for review,
2. Patch is now open for review by other developers,
3. Gerrit triggers a build on Jenkins,
4. Jenkins builds the patch and runs test suite,
5. Jenkins gives the Verified label a score of +1 or -1

[15] Git is a widespread distributed version control system



Reviewer	Code-Review	Verified
Matthias Bohnstedt		
Christian Cikryt		
Franz Zieris		
Felix Dietz		
Ute Neise		
Arsenij Solovjev		
Holger Schmeisky		
Arndt Lasarzik		
Sonarqube QA		
Jenkins CI		✓
Stefan Rossbach	-1	

- Need Code-Review

Dependencies Change depends on 1 outdated change(s) and should be rebased on the latest patch sets.

Subject	Owner	Project	Branch	Updated
Depends On				
[UI] Add another SarosView for browser/html based UI [OUTDATED]	Matthias Bohnstedt	saros	master (Browser based UI)	Dec 2
Needed By				
(None)				

Reference Version: Base

Patch Set 1 d1cbc7bf8cf9e5b1b3d5bb2dbba9b83b9a22adfe
Patch Set 2 82bdb3c18601c62fec064819b7381b774ab56922
Patch Set 3 f250751f8bee8b902e559ec38e99b9be314a9b2c
Patch Set 4 e1ce38de42d3bdb98c0793d213bf8583acb525e1
Patch Set 5 1a5ffedc55b3c6451b06b7aea41b0096e59a8976
Patch Set 6 f546c39ae7c62a51ecce9451d4da488f1d33c033
Patch Set 7 29072bdb86aced7badd8d218307abc213f15aaa4

Author	Matthias Bohnstedt <matthias.bohnstedt@gmail.com> Nov 26, 2014 8:35 PM
Committer	Matthias Bohnstedt <matthias.bohnstedt@gmail.com> Dec 2, 2014 7:44 PM
Parent(s)	1dc8e996c999316762f4991b47289c4a595d7478 [UI] Add another SarosView for browser/html based UI
Download	checkout | pull | cherry-pick | patch | Anonymous HTTP |

```
git fetch http://saros-build.imp.fu-berlin.de/gerrit/saros refs/changes/33/2033/7 && git checkout FETCH_HEAD
```

	File Path	Comments	Size	Diff	
	Commit Message			Side-by-Side	Unified
A	de.fu_berlin.inf.dpp.ui/.classpath		6 lines	Side-by-Side	Unified
A	de.fu_berlin.inf.dpp.ui/.project		10 lines	Side-by-Side	Unified
A	de.fu_berlin.inf.dpp.ui/.settings/org.eclipse.jdt.core.prefs		1 line	Side-by-Side	Unified
A	de.fu_berlin.inf.dpp.ui/META-INF/MANIFEST.MF		7 lines	Side-by-Side	Unified
A	de.fu_berlin.inf.dpp.ui/build.properties		4 lines	Side-by-Side	Unified
A	de.fu_berlin.inf.dpp.ui/resources/html/saros-angular.html		74 lines	Side-by-Side	Unified
A	de.fu_berlin.inf.dpp.ui/resources/readme		2 lines	Side-by-Side	Unified
M	de.fu_berlin.inf.dpp/src/de/fu_berlin/inf/dpp/ui/views/SarosViewBrowserVersion.java	**9 comments**	+43, -9	Side-by-Side	Unified
			+147, -9	All Side-by-Side	All Unified

Comments Expand Recent | Expand All | Collapse All

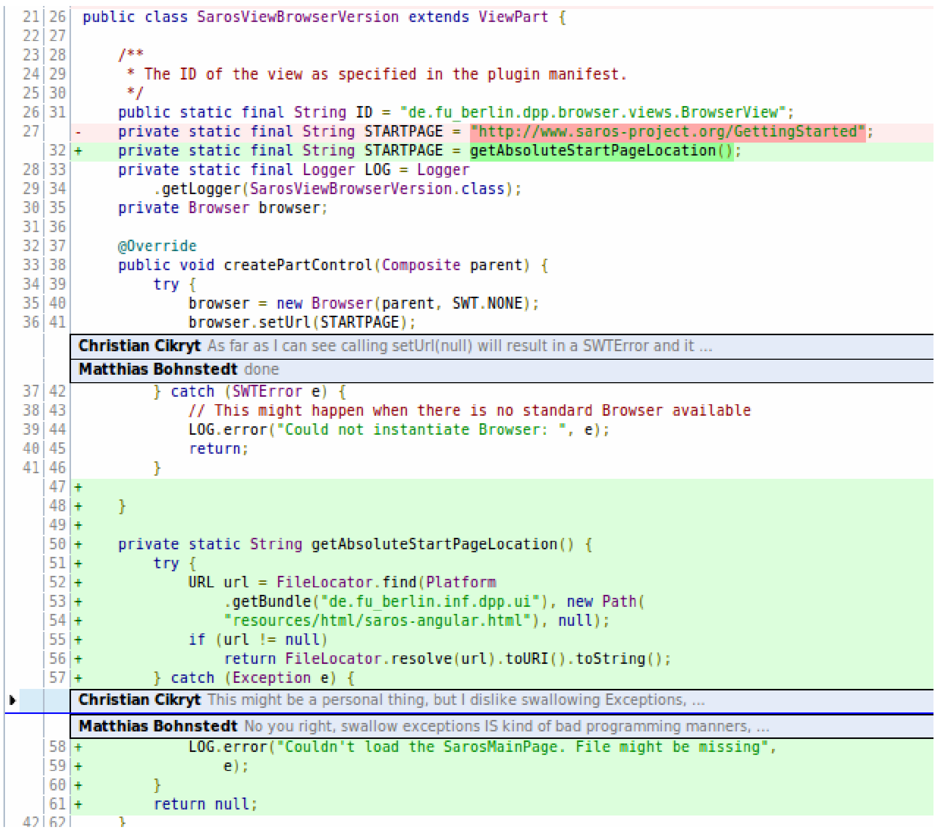
Matthias Bohnstedt Nov 26 8:36 PM
Uploaded patch set 1.

Jenkins CI Nov 26 8:36 PM
Patch Set 1:

Build Started http://saros-build.imp.fu-berlin.de/jenkins/job/Saros-Gerrit-Conditional/720/

Figure 2: Gerrit's web interface for a single change. At the top you see a list of reviewers, below that a series of patch sets. The last patch set is unfolded where a high level structural overview of the patch is given (files changed, how many lines are changed etc.).

6. Reviewers score the patch
    - if two +1 scores are given by two developers, and no negative scores are given, the patch can be merged into the master branch
    - if the patch receives negative scores, the developer is asked to produce a revised patch and the process starts anew.



```
public class SarosViewBrowserVersion extends ViewPart {

    /**
     * The ID of the view as specified in the plugin manifest.
     */
    public static final String ID = "de.fu_berlin.dpp.browser.views.BrowserView";
-   private static final String STARTPAGE = "http://www.saros-project.org/GettingStarted";
+   private static final String STARTPAGE = getAbsoluteStartPageLocation();
    private static final Logger LOG = Logger
        .getLogger(SarosViewBrowserVersion.class);
    private Browser browser;

    @Override
    public void createPartControl(Composite parent) {
        try {
            browser = new Browser(parent, SWT.NONE);
            browser.setUrl(STARTPAGE);
Christian Cikryt As far as I can see calling setUrl(null) will result in a SWTError and it ...
Matthias Bohnstedt done
        } catch (SWTError e) {
            // This might happen when there is no standard Browser available
            LOG.error("Could not instantiate Browser: ", e);
            return;
        }
+
+   }
+
+   private static String getAbsoluteStartPageLocation() {
+       try {
+           URL url = FileLocator.find(Platform
+               .getBundle("de.fu_berlin.inf.dpp.ui"), new Path(
+               "resources/html/saros-angular.html"), null);
+           if (url != null)
+               return FileLocator.resolve(url).toURI().toString();
+       } catch (Exception e) {
Christian Cikryt This might be a personal thing, but I dislike swallowing Exceptions, ...
Matthias Bohnstedt No you right, swallow exceptions IS kind of bad programming manners, ...
+           LOG.error("Couldn't load the SarosMainPage. File might be missing",
+               e);
+       }
+       return null;
    }
```

Figure 3: Gerrit diff view. Here all reviewers can see what changes and where have been made. Also there is a possibility to leave comments for each line.

3.2 Upgrading Saros' infrastructure

Having understood the existing infrastructure in the chapter above, the task was now to find the right way of how to integrate architecture compliance checking into the process. In order to keep with the agile value of not relying heavily on tooling, the most important goal was that the developer doesn't have to install additional tooling. Furthermore, it was important the developer doesn't have to do a lot or any additional work to see the results of the check. If the developers had to navigate to some website, or have do extra work to see the results, it is possible they would not bother to do so after a while. Thirdly, it was important that the feedback from the check is received in a not long interval after the actual submit of a patch. In general the one quality desired from the integration could perhaps be called "seamlessness", meaning that the developer does not notice any structural change in the infrastructure to which he has to adapt (apart from the additional feedback of course).

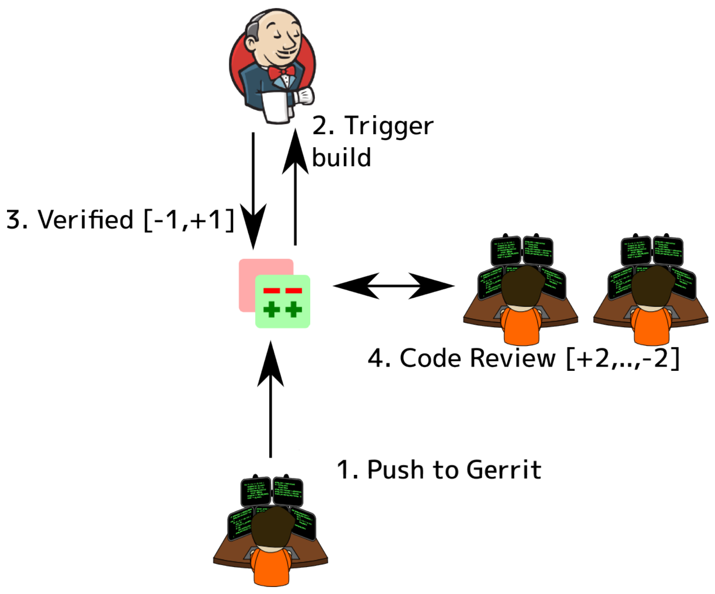


Figure 4: The build/review process in Saros. Icons for Jenkins and Gerrit in this Figure as well as Figure 5 courtesy of http://jenkins-ci.org/ and http://commons.wikimedia.org/wiki/File:Gerrit_icon.svg

Requirement
No installations on behalf of the developer
Ease of access
Quick feedback

Table 1: Requirements for the integration of the architecture compliance check

3.2.1 Sonarqube

Sonarqube is a well-established open-source used by well-known projects such as Apache and Eclipse. It provides the basic set of features I require, such as a an architecture compliance checking mechanism. A Jenkins plugin exists for Sonarqube which eases integration into the development process. Apart from having a lot of the desired features, Sonarqube is open-source, which would allow me to extend it as wished. At this point I had little to no idea how the final solution would look like, that's why it was important that I used a well-established system which had an active community.

As its website states Sonarqube is an "open platform to manage code quality". This means that Sonarqube provides not only static analysis but also a multitude of integration options and views upon the analyzed source code. Providing a complete overview would go beyond the scope of this thesis, so

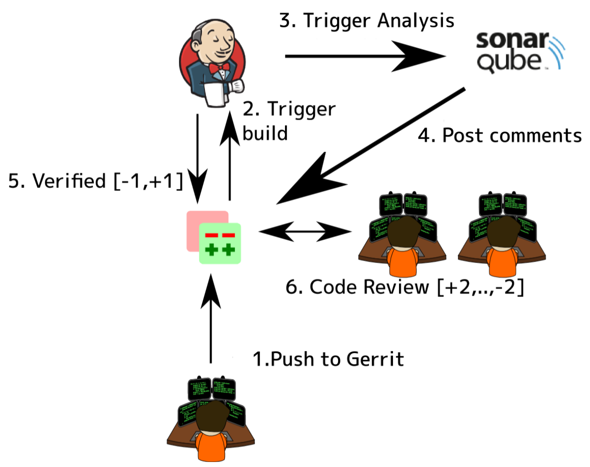


Figure 5: The build/review process in Saros

I will only touch upon the features relevant to this thesis.

The most important concept used in Sonarqube, is a rule. The metaphor for this concept is that it is a rule the project has to comply to. A simple example for a rule would be that every public method has to be documented with Javadoc. Following a rule is enforced with violations (or issues, depending on your perspective). A rule (and thus the issue it raises) has a severity level, which helps the developer evaluate the quality of the code base. An issue has a location, this can be a line of code or a file (there are metrics which are measured by Sonarqube, but this is done with a mechanism which is not discussed in this thesis). If the code base has many critical issues, then the quality is low, if only has minor issues the quality is relatively high. Sonarqube uses profiles to remember a set of customizable rules for each programming language. Such a profile can then be assigned to a project. So, how does Sonarqube know if a project complies to a rule (and by extent to a profile) or not? This is done by running an analysis on the project. Sonarqube provides a general mechanism for analyzing the code base, which plugins are free to use at their own will. Plugins for many well known static analysis tools such as Checkstyle, PMD and FindBugs are provided out-of-the-box.

The rule I intended to use is Architectural Constraint, which is run by the Architectural Rules Engine. In a Sonarqube profile you can define a set of architectural constraints, which specify dependencies between java packages and/or classes.

Taking this Architectural Rules Engine I would upgrade Sonarqube to use a DSL used to describe an architecture. I would use the one developed by Anne Augustin [Aug13], as well as take a look at the ones used in ConQAT and Sonargraph for inspiration. From this point, I would consider what other aspects might be modelled using such a DSL.

3.2.2 Why not Sonarqube's architecture rule engine?

Sonarqube's capabilites in terms of defining architecture rules are limited to Sonar's web interface. This poses an inconvenience, since if a developer would like to make changes to the architecture rules, he would be forced to use yet another tool in an already difficult process. A more preferable approach would be to have a DSL describing the architecture in the source code itself. This way the architecure of the system becomes a programming artifact just like the source code and the JavaDoc documentation (this fulfills the requirement that the developer should have to adapt the as little as possible). As soon it is such, it can be included explicitly into the development process, without the need of opening new communication channels. Rather, the existing infrastructure would be leveraged: one could review and commit changes to the architecture using Gerrit, Git and Jenkins just as one would do with source code and documentation. Moreover, a new feature can include it's own architectural constraints in *one* patch. If passing an architecture compliance check becomes compulsory to merging a patch-set with master, the developer is forced to take the architecture into consideration. However what exact form this would take requires a better analysis of the development process as well as further adaptation efforts by the development team.

Such a DSL has already been developed (and with good results) in Anne Augustine's thesis [Aug13], however her approach is tailored to a specific build infrastructure and cannot be applied to Saros. Also her implementation was impossible to find, as her university email account got discontinued after she finished her thesis and her advisor did not know where the code is.

3.2.3 Integrating Architecture Compliance Checks into the Development Process

There are two artifacts that Sonarqube produces, where the results of an architecture compliance check can be seen

1. The full overview of the technical debt of the project, or
2. an issue report, which shows only the technical debt which was acquired since the last Sonarqube analysis.

3.2.4 Issue reports

Sonarqube provides an Issues Report Plugin. This plugin provides a report (currently as plaintext or html). The Issues Reports Plugin requires Sonarqube to analyze in incremental mode. What this is means, is that Sonarqube looks for the files which have been modified since the last analysis, and analyzes only those files. A comparison to the default full analysis can be seen in the table below.

	Incremental	Full
Files analyzed	Only those which have been changed since the last full analysis	All
Duration of a Saros analysis	2 minutes	15 minutes

Table 2: Comparison of incremental and full analysis in Sonarqube

As you can see from the table, the incremental mode is preferrable because, firstly, it can give faster feedback, secondly, it has a higher ratio of relevant issues. One problem is that incremental mode was meant to be run locally (according to discussions on the Sonarqube forums), for a developer to get quick feedback on his work in progress and as such the issues report plugin was made version history agnostic. That means that the diff for which files to check simply takes the last Sonarqube analysis as reference. So if one has a central server, which runs Sonarqube, then the increment for the issues report is not necessarily between a commit and it's parent commit, but between a commit and that commit which was previously analyzed. When using a distributed SCM such as Git, that previous commit is generally not the parent commit.

This poses a problem, as the incremental analysis analyzes too many files, the issues report is bloated with issues from files which presumably don't interest the developer. Assuming that such a bloat would lessen the interest in reading the issues report it was necessary to limit the range of such an issues report.

To resolve this problem I decided to write a script that would run with each Jenkins build for a new patch. There were two ideas on how the script should integrate issue reports into the development process. The first one failed and the second didn't, both are elaborated upon in the sections below

3.2.5 Failed first approach at integrating Sonarqube with Gerrit and Jenkins

To amend this I decided to run a Sonarqube analysis on the parent commit explicitly before running it on the current commit. Another option would have been to fork Sonarqube and to implement a new mode which is version-history aware. However this appered to be to cumbersome to implement and I sought other options.

To save time needed to obtain and build the previous commit (Sonarqube requires both source and executable), I decided to archive potential parent commits. Before a usual build in Jenkins there is a new step: a script is run which finds the archived build of the parent commit, and runs Sonarqube on that commit. This way the next analysis Sonarqube performs will only analyse the files that have changed between the parent commit and the current commit, which is exactly what we wanted.

This leads us to a problem however, if we archive every commit, we will eventually simply run out of disk space. So to prevent this two approaches were considered on how to reduce the number of archived commits.

1. In the first approach I simply set an expiration date for every commit, and configure a job on Jenkins which deletes expired archives and is run periodically (eg. once a week). The upside of this is that it is easy to implement, the downside however is that some archives might be missing when they are required. This is only slightly inconvenient, since we can always reset to the parent commit, *build* it, and then run an analysis. This however would make a build on Jenkins last twice as long, which is undesirable.
2. Determine a more complicated criteria on how to determine which commits *can* be a parent to some commit in the current development. One approach would be to query Gerrit, which has a list of open patches. Each patch could be queried for it's parent commit and add it to a set of possible parent commits. Then one could see which of the archived patches are in this set, keep those, and delete the rest.

I chose to write this is script in Lua, because it was already in use for scripting tasks in Saros' infrastructure. Also I used the opportunity to learn a new programming language and a new paradigm. Lua's philosophy grounds in minimalism, the language itself doesn't have many constructs and prides itself in having a really small compiler. This allows for good usage with embedded devices.

I wrote a script which archives a parent commit, and executes a Sonarqube run on an archived commit, however, the problem remained how to determine the parent of a commit. However this problem became inconsequential, since midway through the development I found out that this approach cannot possibly work. The reason was that when Sonarqube performs an incremental run, it is in increment of the last *full* Sonarqube analysis, not the last incremental analysis.

3.2.6 Successful second approach at integration

After the failed approach presented in the section above, the problem arose of how to still be able to perform incremental analyses, when a prior full analysis is required.

Now let us reconsider the whole problem from a bird's eye view. A simple solution is to run a full analysis on the parent commit, for each new patch. The problem is that it takes a lot of time for Saros (around twenty minutes), which was not agreeable to my supervisor. An even simpler (and worse) solution would be to not have a full analysis run before. Although this would run faster, the problem is that the increment would be done from a very old analysis of Sonarqube and would contain many files which the developer hasn't even edited. The issue report produced thusly would again have a very small ratio of relevant information to the developer.

So, what to do? As is the usual practice in software development, if finding an exact solution to a problem is not viable, a heuristic is used. Upon a hint from my supervisor, a heuristic with the following steps was developed:

- A full Sonarqube analysis is run on Saros' master branch every night (as not to consume computation power, which is used during the day for other tasks).
- An incremental analysis is run and produces an issue report, incremental to yesterdays master branch.
- A script gathers commit information from Git, and filters out irrelevant issues.
- These issues are then posted to Gerrit via Gerrit's REST API as inline comments to the diffs they affect.

This solution fully satisfies the requirements in table 1.

- the time needed for a report is reduced by having Sonarqube analyse only the diff between yesterdays master branch and todays commit,
- And the particularly nice part of this solution is that the developer doesn't have to change his workflow at all. The only change is that there is an additional reviewer to his patches

3.2.7 Implementation of the second approach

I wrote the soution as two Scala scripts. Beside personal taste I chose Scala because it provides a very compact syntax for parsing XML document. This was insofar nice, as issue reports come in an html format.

The first script `gitClient` can query Git for the repository it is in. I used it to retrieve information about what files are in a commit, what subprojects the commit affects[16] and whether the commit was a merge-commit.

The second script `sonarReview` receives an id for a build using which it can find a previously generated issue report. The issue report is then parsed into a message format that Gerrit can understand[17] and posts the message to Gerrit. During parsing issues which are not in a file that was part of the commit (this is found out by a call to `gitClient`) are discarded, so that only relevant issues are posted to Gerrit.

The message is then interpreted as a patch review by Gerrit. The forthcoming thing about Gerrit here is that it allows to post comments to single lines in the diffs. As you remember, issues in Sonarqube usually pertain to a single source code line. This aspect was readily used, and each issue found in the issue report was made into an inline comment in the Gerrit review. This way Sonarqube becomes just another reviewer on Gerrit, which is very nice, since from the developers point of view not much changes.

3.2.8 Jenkins Configuration

Having the working scripts, a series of configurations on Jenkins had to be done. This included the full Sonarqube analysis mentioned above called `Saros-Full-Sonar-Nightly`. Also the job `Saros-Gerrit`that is triggered by Gerrit every time a patch is pushed for review had to undergo big changes.

`Saros-Gerrit` built and ran tests on the complete Saros project. This posed a problem, as a Sonarqube analysis could be run only on a subproject. This meant that a Sonarqube analysis should have been run for each subproject on build. This took a long time to execute (over thirty minutes) and was not accepted by the team. An optimization was due.

The simplest way to optimize the build was of course to only run a Sonarqube analysis only on those subprojects which are affected by the last commit. This was not easy to implement, as it required finegrained configuration of

[16]as you will see further, this is important for the job configuration on Jenkins
[17]JSON

several jobs. The cognitive effort required was akin to that of programming a task of medium difficulty. However when one is programming, one can code for a while, then compile (or have an instant compile check by the IDE) to receive feedback if the code is at least syntactically correct. When confguring Jenkins one does not have that luxury, to see if the configuration works, one has to retrigger a build and see the results (which did take a little over a quarter of an hour). This in stark contrast to programming, where the feedback cycle from the check is mere seconds.

The final configuration includes a new job that superceded `Saros-Gerrit` called `Saros-Gerrit-Conditional`, as well as Sonarqube (or QA) jobs for each of the four Saros subprojects. The interaction is like this:

1. Gerrit triggers a build
2. `Saros-Gerrit-Conditional` builds the complete Saros project
3. Using `gitClient` the job queries what subprojects have been affected by the and calls the QA jobs for those subprojects.
4. The workspace including the generated executables is copied to those jobs using `rsync`[18] and the jobs are started
5. each such job runs unit tests and a Sonarqube incremental analysis which produces an issue report
6. `sonarReview` is called to post the issue report as a review to Gerrit.

3.2.9 Custom submit rules

There was some discussion with my supervisor about whether Sonarqube should be able to score the patches. Indeed the original intent was to let Sonarqube affect what can or cannot be submitted to the master branch. The Gerrit way to customize this is to define the score criteria necessary to submit a patch to the master branch[19] by writing a custom submit rule[20].

However it turned out, that it wasn't possible to implement a rule that fulfilled our criteria. We wanted to make the users Sonarqube and Jenkins both use the label `Verified`. A patch would be submittable if both Jenkins and Sonarqube had scored the patch a +1. The problem was this: if a patch

[18]This is an optimisation, otherwise the QA jobs would have to build the executables again

[19]As you remember the default criteria was an aggregated score of +2 for Code-Reviews and +1 for Verified.

[20]Details on this mechanism are documented here: https://gerrit-review.googlesource.com/Documentation/prolog-cookbook.html#HowToWriteSubmitRules

has a score of `Verified` +1 how do we know that it has been scored by both systems? My attempt at solving this problem was trying to define a rule which would check whether both Sonarqube and Jenkins have scored the patch and whether the `Verified` label had a score of +1. The former proved to be impossible given the tools Gerrit provides to accomplish the task, as one cannot query the reviewers of a patch and what score they have given, if any.

4 Implementing Architecture Compliance Checking

The plan was to first implement an architecture compliance checker which would do a simple check whether the source code corresponds to a layer definition, and then proceed from that.

The first step was of course to find a suitable framework, the particulars of which will be given below. Following this will be a more detailed elaboration on the selected technology as well as an iteration-for-iteration breakdown of the development of the DSL it's tooling.

4.1 Evaluation of static analysis tools

From the beginning of this thesis it was apparent what basic requirements the technology underlying the architecture compliance check would have to meet.

1. It had to provide some sort of means to analyze source code on a high level class and package organization basis, as well as enable me to get into the particulars of class, from method calls to variable declarations.

2. The underlying technology had to be open-source, not only because of monetary interest, but also in order to keep in line with the spirit of Saros, which is released under a GNU licence. Also a faint hope, that this thesis would have relevance beyond the Saros project, was at play for this decision.

3. Whatever the implementation of the rule was it had to be able to run as Sonar plugin. This is mostly a historical decision, since so much had already been dedicated to integrating Sonarqube with Jenkins and Gerrit. What helps is the fact that Sonarqube is a well established open-source product and could provide a distribution platform.

4. It was not yet determined what aspects the DSL would describe. It was possible that the DSL would only describe layers, but the intent was to encompass more architectural aspects. As such there was need to be able to generate different types of Sonarqube issues depending on context.

Requirement	Severity
Detailed source code analysis capabilities	Indismissible
Open-source	Preferred
Ease of integration with Sonarqube	Indismissible (at this point)
Multiple issues	Preferred

Table 3: Requirements for the underlying technology for the architecture compliance check

4.1.1 NDepend and others

The first approach that came to mind was to follow the approach that Anne Augustin implemented. In her thesis, [Aug13], she was parsing an architecture definition, written in a custom DSL called Archibald, into a series of code queries[21], which are then used to verify if the source code complies to the constraints defined in the DSL. The code query engine she used is NDepend, which is native to the .NET framework.

Since Saros is an open-source project based on open-source technologies, using .NET was out of the question[22]. The same applies to other tools such as ConQAT, Sonargraph and jQassistant.

4.1.2 JArchitect

The alternative to NDepend Augustin suggested was called JArchitect[23]. JArchitect provides free licences for open-source projects. The people behind JArchitect promptly replied to my request for a licence. However upon inspecting the tool closer upfront, I found out that it provides no API for executing code queries, one can only execute code queries through the GUI of the program. As such JArchitect contained the necessary capabilities, but didn't provide means to use it in a way that was necessary.

4.1.3 PMD

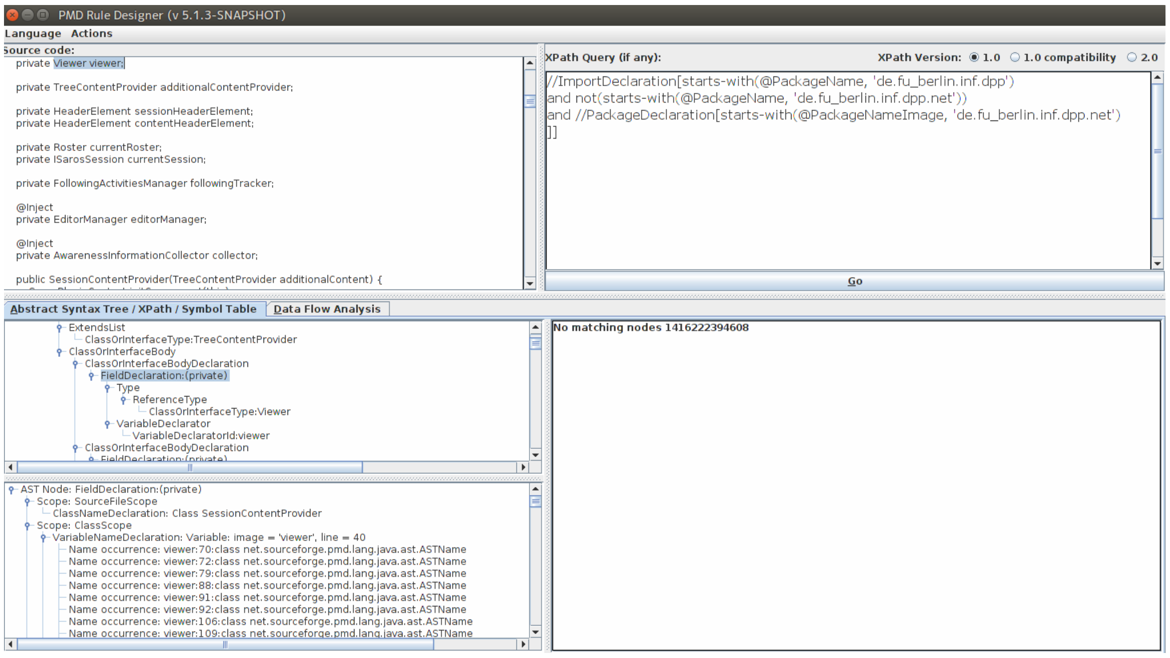
Upon revisiting the problem of finding a suitable technology for architecture compliance checking I stumbled upon PMD, which was already shipped with previous versions of Sonarqube. PMD analyzes the Java abstract syntax tree

[21] In the sense of Code Query Language, where code is treated as data in a database, and can be queried over in a language with similar goals in mind as SQL

[22] Admittedly, since the time of this evaluation the .NET Platform has been announced to become completely open source. See http://www.heise.de/developer/meldung/Microsoft-NET-wird-komplett-Open-Source-2452033.html, link verified on 24.11.2014

[23] http://www.jarchitect.com/

(AST)[24] of a program and can execute queries upon it. So now I could attempt and define an architectural constraint for the architectural violation found in **??** by analyzing the AST. PMD allows to define rules using XPath on the AST and also provides a convenient GUI-Tool, which is shown in Figure 6, to test and develop these. The rule was promptly defined and Sonarqube correctly identified the offending class and raised the appropriate architecture violation issue.



Source code:
```
private Viewer viewer;

private TreeContentProvider additionalContentProvider;

private HeaderElement sessionHeaderElement;
private HeaderElement contentHeaderElement;

private Roster currentRoster;
private ISarosSession currentSession;

private FollowingActivitiesManager followingTracker;

@Inject
private EditorManager editorManager;

@Inject
private AwarenessInformationCollector collector;

public SessionContentProvider(TreeContentProvider additionalContent) {
```

XPath Query (if any): XPath Version: [✓ 1.0] 1.0 compatibility 2.0
```
//ImportDeclaration[starts-with(@PackageName, 'de.fu_berlin.inf.dpp')
and not(starts-with(@PackageName, 'de.fu_berlin.inf.dpp.net'))
and //PackageDeclaration[starts-with(@PackageNameImage, 'de.fu_berlin.inf.dpp.net')
]]
```

Abstract Syntax Tree / XPath / Symbol Table | Data Flow Analysis
```
ExtendsList
  ClassOrInterfaceType:TreeContentProvider
ClassOrInterfaceBody
  ClassOrInterfaceBodyDeclaration
    FieldDeclaration:(private)
      Type
        ReferenceType
          ClassOrInterfaceType:Viewer
      VariableDeclarator
        VariableDeclaratorId:viewer
  ClassOrInterfaceBodyDeclaration
    FieldDeclaration:(private)
```

```
AST Node: FieldDeclaration:(private)
  Scope: SourceFileScope
    ClassNameDeclaration: Class SessionContentProvider
  Scope: ClassScope
    VariableNameDeclaration: Variable: image = 'viewer', line = 40
      Name occurrence: viewer:70:class net.sourceforge.pmd.lang.java.ast.ASTName
      Name occurrence: viewer:72:class net.sourceforge.pmd.lang.java.ast.ASTName
      Name occurrence: viewer:79:class net.sourceforge.pmd.lang.java.ast.ASTName
      Name occurrence: viewer:88:class net.sourceforge.pmd.lang.java.ast.ASTName
      Name occurrence: viewer:91:class net.sourceforge.pmd.lang.java.ast.ASTName
      Name occurrence: viewer:92:class net.sourceforge.pmd.lang.java.ast.ASTName
      Name occurrence: viewer:106:class net.sourceforge.pmd.lang.java.ast.ASTName
      Name occurrence: viewer:109:class net.sourceforge.pmd.lang.java.ast.ASTName
```

```
No matching nodes 1416222394608
```

Figure 6: The PMD Rule Designer allows the definition of XPath queries (upper-right window), that can be executed on a piece of Java code (upper-left window). For better comprehension one can additionally see the AST of the source code in lower left window.

Having reassured myself that PMD is the adequate tool for the task, I set out to find a way how to load rule definitons in PMD dynamically. I wanted to be able to load rules dynamically as it would allow for greater extensibility. Each constraint the DSL imposes would be translated into it's own rule. This has the advantage that each rule couldhave it's own severity and error message. To my disappointment however it appeared that there is no way to load XPath rules (my preferred approach) dynamically. At this point it would be helpful to explain the inner workings of PMD and how to write rules for PMD in the alternative way, namely in Java.

[24] according to Wikipedia, an AST is a tree representation of the abstract syntactic structure of source code written in a programming language

4.1.4 Technical elaboration on PMD's viability

A new rule in PMD written in Java extends `AbstractJavaRule` and implements the polymorphic method `visit(ASTNode)`, where `ASTNode` is any kind of ASTElement. Such an extension can conveniently add a violation to the PMD report which later translates to a Sonarqube issue. A Sonarqube issue is what the developer will ultimately see as a Gerrit comment.

A rule is visible to PMD if it is declared in a ruleset file. The ruleset file declares a rule name and an implementing class (among other things such as description, severity etc.).

Moreover the Sonarqube-PMD plugin uses several files to configure a given rule:

- `pmd.properties` this is a file used for localization. It defines a property called `rule.pmd.<rulename>.name`. It is here that Sonarqube looks up the name of the rule.
- `rules.xml`. This file specifies where the file is defined in PMD, more specifically it maps a rule to the configuration file which defines it originally in PMD.
- a `<rulename>.html` file which contains the description for the given rule, which will be displayed in the documentation for the rule.

To execute the rule, Sonarqube simply needs to have the compiled plugin added to its `plugins` folder and be restarted. The rule can then be added to a quality profile and the next analysis will evaluate the rule.

During an analysis the `pmd.xml` is copied to the `.sonar` folder inside the projects root folder and is used during a Sonarqube analysis, which is also run in the same folder.

All of these files are modifiable during runtime. Theoretically this would enable creating a rule for every aspect the DSL covers at runtime, which was my first approach.

4.2 Approaches to Implementation

4.2.1 Create rules from DSL dynamically

At first I followed the idea of creating a rule for each type of architecture violation dynamically (in hindsight, this was not so smart, since these were still unknown). This required a more detailed understanding of how exactly `sonar-pmd` calls PMD-rules. To this end I created a series of sequence diagrams (Figures 7, 8 and 9), which follow the method calls starting from the invocation of the `sonar-pmd` plugin to the points where the rule definition is loaded, as well as the point where the rule is executed.

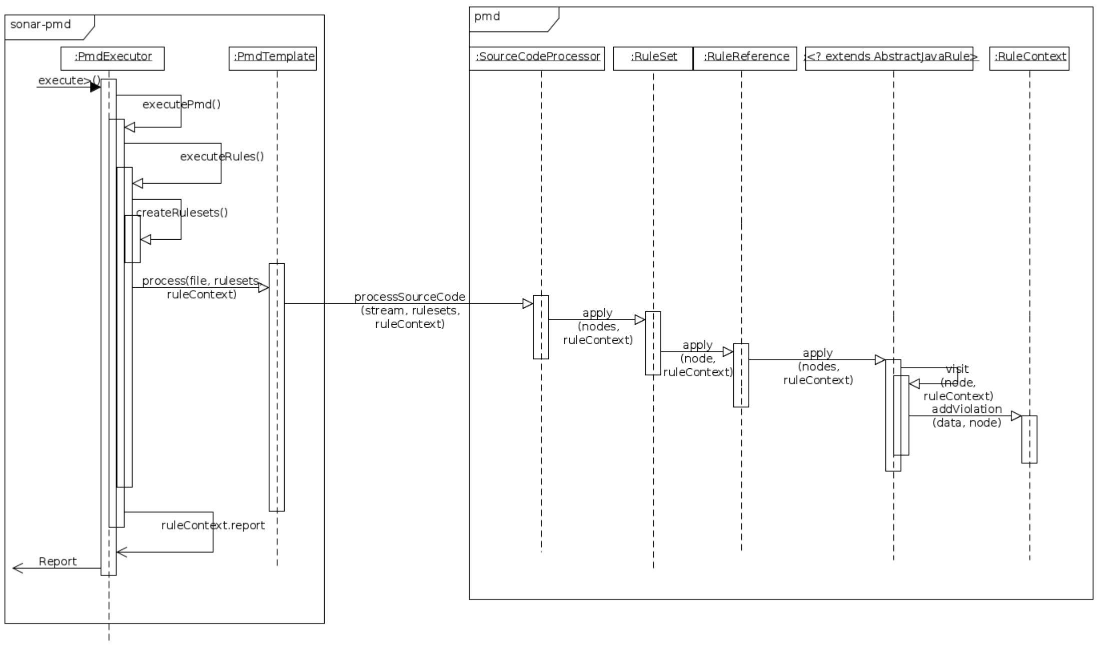

sonar-pmd
:PmdExecutor
:PmdTemplate
execute()
executePmd()
executeRules()
createRulesets()
process(file, rulesets, ruleContext)
ruleContext.report
Report
pmd
:SourceCodeProcessor
:RuleSet
:RuleReference
:<? extends AbstractJavaRule>
:RuleContext
processSourceCode (stream, rulesets, ruleContext)
apply (nodes, ruleContext)
apply (node, ruleContext)
apply (nodes, ruleContext)
visit (node, ruleContext)
addViolation (data, node)

Figure 7: General overview. A sequence diagram showing the calls between an invocation of the plugin (which calls execute() on PmdExecutor). The rules are loaded during the call to createRulesets(). PMD is invoked during the processSourceCode(). PMD receives the stream holding the source code file, the set of rules and an object it shares with sonar-pmd called the RuleContext. When a PMD-rule discovers a violation, this violation is added to the RuleContext. Later sonar-pmd uses the RuleContext as a basis to raise issues in Sonarqube. PMD goes through the set of rules, (which contains only metadata, such as name and class, but no logic), loads the corresponding rule implementation in RuleReference. Subsequently visit() is called on each rule defined in the set of rules. This where the actual logic of a rule is executed.

Having researched the mechanism of rule loading and execution in `sonar-pmd`, I was able to pinpoint the location where I would be able to add dynamic rule creation and execution. This would have required to:

1. analysing the DSL and creating the rules dynamically before `createRulesets()`

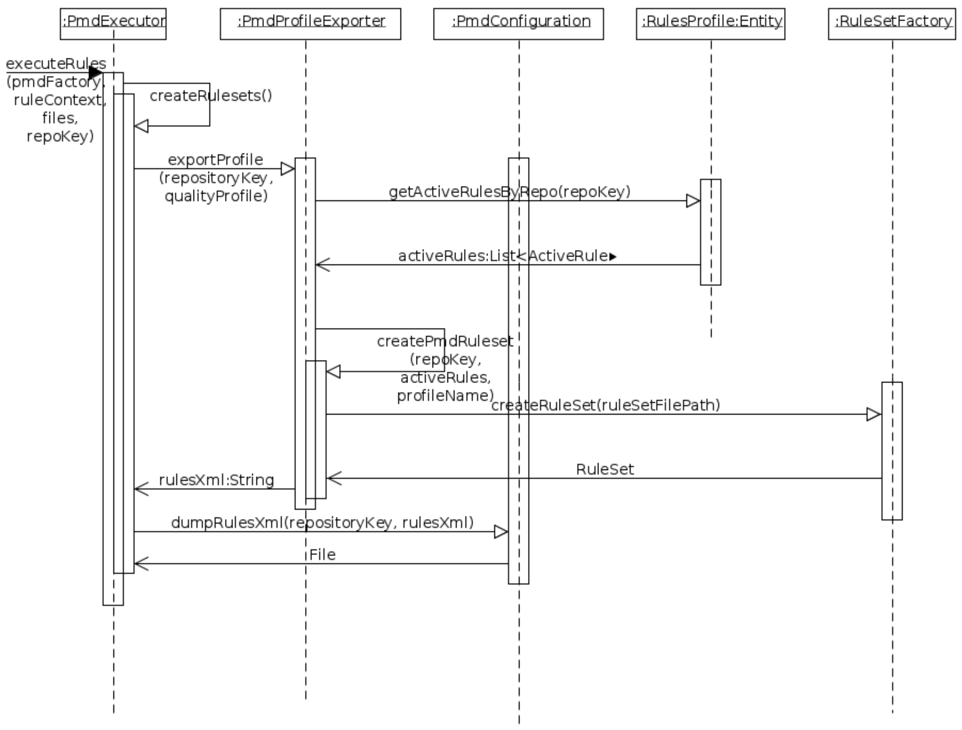

:PmdExecutor
:PmdProfileExporter
:PmdConfiguration
:RulesProfile:Entity
:RuleSetFactory
executeRules (pmdFactory, ruleContext, files, repoKey)
createRulesets()
exportProfile (repositoryKey, qualityProfile)
getActiveRulesByRepo(repoKey)
activeRules:List<ActiveRule>
createPmdRuleset (repoKey, activeRules, profileName)
createRuleSet(ruleSetFilePath)
RuleSet
rulesXml:String
dumpRulesXml(repositoryKey, rulesXml)
File

Figure 8: A closer look at createRulesets(). During this call sonar-pmd fetches the active rules defined in the Sonarqube profile from RulesProfile. From these a RuleSet is created by the RuleSetFactory. This RuleSet is then dumped as an .xml in the root of the project where the analysis is being executed.

is called.

2. adding the newly created rules to the RuleSet.
3. making sure the RuleSetFactory was able to load the dynamic rules at runtime.

Since point 3. seemed the riskiest of the three, this was what I attempted to do first. I had written a class that can generate a class which extends `AbstractJavaRule` and has a `visit()` which analysed whether a class in the network package accessed members of other Saros packages [25]. This worked nicely when run as a standalone unit test. However I was not able to generate the class at runtime during the Sonarqube analysis. At that

[25]This was an exemplary architecture problem I knew of in Saros. Some classes in the network layer used business logic which they shouldn't have.

point following this approach became too risky. Luckily, another approach dawned.

4.2.2 Use one static rule

One of my most productive days was throwing away 1,000 lines of code.

Ken Thompson

After a few unsuccesful attempts at generating rules dynamically, and upon wise advice from my supervisor, I decided to implement a rule which rather than representing a single architectural rule, considers the complete DSL in one go. That means each source file would be checked on whether it adheres to the DSL. For this the following elements were required:

- The DSL itself
- Saros' architecture described in the DSL
- A PMD rule which would be able to detect compliance breaks between the DSL and the source code

4.3 Iteration Zero

I followed an iterative approach, and this were the outlined steps for the development portion of an iteration. The complete iteration would encompass the following steps:

- Requirement elicitation:
  - what aspects of the architecture to model next
  - technical issues
- Development: the implementation itself
- Evaluation: Deploying the DSL and PMD Plugin to the productive CI-Environment in order to receive feedback from the Saros team which would be incorporated in the next iteration.

When the technical side of development would be set, I would have a framework in which I could further refine the architecture compliance checking. The next step would be to determine what aspect of the architecture one can reasonably model.

In the iteration zero my goal was to write a basic DSL, which defines components and which components can access which component. The further challenge of this iteration was to establish a turnaround cycle from conception of the DSL through to its use in the Saros QA cycle.

First the grammar for the DSL (for the time being called Archnemesis) was written.
As you can see in Figure 10, a number of elements are defined, allow to elaborate on their semantics here briefly:

Component has a unique name, and a Java package which corresponds to it.

RootComponent is a component of which all components in the system are subcomponents. More specifically, it's `packageName` feature is used to discern whether some dependency belongs to the system we describe, or to a third party library.

Connector A connector consists of two component names connected by a right- or double-arrow.

> **UniDirectional** connectors specify that the component `from` can access members from the component `to`, but not vice versa.
>
> **BiDirectional** connectors specify that two components can access each others members.
>
> If two components are defined but not in a connector relationship, then they cannot access each others members. All packages not represented in the DSL are not under any constraint.

In Figure 12 you see a literal transcription of the component diagram **??** made by Patrick Schlott [**?**, Schlott2013]

There are a few differences between the diagram and the DSL. “Activity Providers” has been omitted as it seemed to require more precise modelling which was to be done at a later point and was technically already included in the package “Session Management”. “Shared Session Data” has also been merged into “Session Management”. Also in the diagram there are arrows going from “Network Layer”, these however are simply a depiction of static dependencies between components in the current state and not the desired state(network should not use any business logic). The same applies to the dependencies between “User Interface” and “Invitation”, and “Session Management” and “User Interface”. Also “Concurrency Management” has been renamed to `operational_transformation` since it is a more unambiguous term (“concurrency” and “management” both mean very general things).

Having completed the first version of Archnemesis I could now let the Xtext framework generate a number of artifacts from the grammar, in particular a parser. This parser could then be used by the PMD rule I wrote to parse the `architecture.chralx` and see whether the source code complies to the description.

This was more time-consuming than it sounds. First I had to convert the Xtext project generated by Eclipse (this is the default use-case for Xtext) to a Maven project. Now it is necessary to highlight some aspects of Maven. Maven is a release management tool, used for building, testing, publishing, generating documentation and various other tasks. The aspect where Maven is different from previous build tools (such as `make` or `ant`) is that it does automatic dependency management. For this there is a configuration file declaring the dependencies of the given project, and Maven does the job of downloading those dependencies at build time from a default or preconfigured repository. Maven can also publish to a repository (the default being the local repository), and make the given project available to other Maven projects. I needed Maven to publish my Archnemesis grammar project to my local repository, so that I could use it as a dependency in the PMD project. Upon some trial and error this was achieved.

The next challenge was to write the logic on parsing the DSL. I successfully followed an approach proposed on a certain blog post[26]. However the object tree(an object represntation of the DSL) I was receiving upon parsing did not have its references resolved. If you take a look at the grammar, specifically these lines,

```
UniDirectional: from=[Component] '=>' to=[Component
    ];

BiDirectional: comp1=[Component] '<=>' comp2=[
    Component];
```

you will notice the square bracket syntax. This syntax denotes cross-references, and for some reason these could not be resolved. After a couple of attempts of resolving the problem myself I tried contacting the forums and did receive a prompt response[27]. Having tried the proposed solution I was successful (frankly, I'm still not sure what the problem exactly was).

At this point all the technical obstacles had been removed, the modified PMD plugin had been deployed to `saros-build`, the team was notified of the introduction of Archnemesis into the project and was prompted for feedback and suggestions, and I was able to continue refining the DSL and

[26]http://davehofmann.de/blog/?cat=22

[27]you can see the full thread here: https://www.eclipse.org/forums/index.php/t/841708/

the description.

4.4 First Iteration

4.4.1 Requirement Elicitation

After the the project had been presented to the Saros team, some feedback came back. There were a couple of ideas voiced (each is provided with a label):

Constraint One of the developers notified me of a problem she had often encountered while working on her feature. She would use the `ui` component, however she would forget to call this component from within the SWT thread.

Grouping She also suggested a that a grouping of components could be helpful, so that one could define the architecture in a similar manner that MVC is defined.

Literate My supervisor suggested adding literate programming[28] to the DSL for greater comprehensibility.

Multipackage A developer noted that classes responsible that are part of the "Invitation" component were making calls to "User Interface", which was making his work very difficult. As a background, his project was porting Saros to Intellij IDEA [29]. This encompassed decoupling the Saros core from the user interface which was Eclipse specific.

"Constraint" posed an interesting challenge as it required a new type of element in the DSL, namely a constraint. This constraint would express the notion that "all access to the `ui` component should be done from within the SWT thread".

Since this could potentionally point to a lot of false positives, I deemed it necessary to be able to adjust the error message which would be seen in the comments, in order not to let all the false positives trivialize the impact of the issue.

PMD in concert with Sonarqube does not support multiple issue messages for one rule (although a method is provided by PMD which allows to provide a customized error message, this was not respected by Sonarqube during a test run).

[28]http://en.wikipedia.org/wiki/Literate$_{p}rogramming$

[29]Intellij IDEA is a well-known IDE

Luckily, Sonarqube provided its own API for writing AST-based rules, which allows for custom issue messages. So it was worth a consideration. There was also another point why migrating to the Sonarqube AST was good: it would make publishing the project on GitHub[30] easier.

The issue was that up until point I had very naively worked on a fork of PMD. The fork contained all of PMD's source code, including all rules for all languages. Although PMD allows to separate a single rule into a single project (for it to be executed it simply must be on the classpath of PMD when it is run), the effort for this seemed on par with a migration to Sonarqube's AST.

Thus, the first goal of the iteration was to migrate from PMD to Sonarqube's AST.

The second goal comes from the problem voiced by one of the developers concerning an undesirable access from the "Invitation" component to the "User Interface" component. As you can see in the first version of the chralx file (Figure 12), the `invitation` component does not have access to `ui`. So what was the problem? The offending class in this case was not inside the package `de.fu_berlin.inf.dpp.negotiation`, but in `de.fu_berlin.inf.dpp.invitation`. This was the case because the package name had different namesClearly, the `invitation` component should have entailed both. This was also the case with the packages `.session` and `.project` which were still in a phase of refactoring (the `.project` package became deprecated). So it was clear that a component should be able to entail multiple packages.

The third goal was to simply add literate programming to the DSL, as the idea seemed really sound. To sum up these were the goals set for the iteration:

1. Migrate from PMD to Sonarqube's AST
2. Allow components to represet multiple packages
3. Literate programming

The other ideas, constraint and grouping were left for the next iteration as they seemed to complicated to implement at first.

[30]A web-based hosting service on which many open-source projects are hosted. Users can create, fork and contribute to open-source projects freely using this platform.

4.4.2 Development

After the requirements elicitation, presented in the previous subsubsection, the development efforts started. At first it was important to evaluate Sonarqube's AST technology.

Sonarqube provides a project on GitHub called `java-custom-rule`. This project implements a couple of dummy rules, and serves to illustrate how implement custom java rules for Sonarqube. To evaluate the rule the packaged `.jar` has to be simply deployed to the `plugins` folder of Sonarqube, and enabled in the quality profile in Sonarqube's web interface.

Difficulties arose however as I tried to reimplement the logic of the rule, which was already written using PMD's API, using the Sonarqube API. Indeed, to my surprise and after a good period of trial and error, I had come to the conclusion that surprisingly the API has no direct means to access the name of a token[31]. Once again contacting the community proved to be helpful, as a response came quickly on the Sonarqube developer mailing list when I wrote my inquiry.

A developer sent a link to his own projects on GitHub where he had solved the same problem. As it turned out, to obtain the name of a package or an import, one has to concatenate the `identifiers` of several AST elements which are linked to each other.

After this problem was solved the migration went pretty straightforward. I created a new project called `archnemesis-sonar-rule`, which can now be found on Saros' GitHub home page.

Implementing multipackages was pretty simple, I had to simply change the rule for a component from

```
Component: 'component' name=ID ':'namespace=
    PackageName ;
```

to

```
Component: 'component' name=ID ('+=' namespaces+=
    PackageName)+ ;
```

After this change the symbol was changed to `+=`, the compound assignment operator in Java, to reflect that a component is a sum of packages in a sense. The `+` at the end is a cardinality symbol, which means that a component

[31]For example "name of a token" in the case of a package declaration would refer to the fully qualified package name, or in the case of a variable declaration to the name of variable

can have one or more namespaces.

To implement literate programming, I first consulted Wikipedia to get a general idea. The important thing I found out, is that a literate program consists of two parts: the essay and the tangled code. The essay is usually written in simple plaintext (or in LaTeX, as is the case with Literate Haskell), without any special syntax to signify that this is the essay portion of the program. The tangled code however is usually written in a block construct, which tells the compiler to handle everything within the block as program code.

My first naive attempt was to have a prefix for tangled code, and have a grammar rule which covers everything which isn't using the prefix. This idea seemed so simple to implement, however it took a good amount of trial and error (which might have been avoided had I had a better understanding of the subject) to see that this was not so simple to do. There were many ways to define a grammar rule which matches everything but a certain type of something. However most of them didn't compile when I ran Xtext to generate the parser and other tooling. And those that did compile produced some unexpected results. The main problem was, however that the grammar rule overshadowed other native rules (such as ID, which is any kind of identifier). The easy way out would have been to implement the essay rule as a multiline comment, however this was a unelegant solution as it went against the notion of literate programming, that the essay part should be a first class citizen.

Ultimately, I settled to having every component and connector declaration prefixed with a "`-- `" and have the essay part be everything that starts with a capital letter and ends in with a punctuation mark. That last aspect might even be preferrable than having the essay portion of the program code be in a free format, as it forces the developer to adhere to a form.

Thus the entry rules for the grammar were changed from

```
Architecture: rootComponent=RootComponent (
    components+=Component)+
(connectors+=Connectors)+ ;

Element: Component|Connector|Constraint ;

...
```

to

```
Architecture: TANGLE_PREFIX rootComponent=
    RootComponent
(elements+=Element)* ;

Element: {Element}(tangle=Tangle)|essay=Essay;

Tangle: TANGLE_PREFIX(component=Component |
    connector=Connector |
constraint=Constraint) ; Essay:
    PROPER_ENGLISH_SENTENCE ;

terminal PROPER_ENGLISH_SENTENCE: ('A'..'Z'|'1'..'9'
    |'*')-> ('. '|'!
'|'? '|'\n\n'|'\r\r') ;

terminal TANGLE_PREFIX: '-- ';

...
```

As you can see now, a the DSL now consists of a root element (with a tangle prefix) followed by an arbitrary number of elements. An element can be either tangled code, or an essay. Tangled code is always prefixed with "`-- `" and is either a component, connector or a constraint. An essay consists of proper English sentences. A `proper_english_sentence` starts either with a capital letter, a number or a * (the latter allows for lists), and ends with a punctuation mark which ends a sentence in English or an end of a paragraph (two line breaks).

After this the `architecture.chralx` was rewritten to adhere to the syntax. Also the description was made in the style of literate programming, meaning that before each component declaration its explanation was written in the essay part. Most of the description is overtaken and translated from [**?**, Schlott2013] You can see this version of the file in the Appendix.

Another change was that now the `architecture.chralx` did not contain an explanation of the syntax itself any longer. Instead the explanation was moved to a separate file called `primer.chralx` which explains and exemplifies the language. It can also be found in the Appendix.

4.4.3 Evaluation

Since both chralx files were part of the projects source code, these were simply submitted in a patch to Gerrit. Some comments came from the team, however these were just seeking some clarifications.

4.5 Second Iteration

This was the last iteration I was going to do, so I decided to keep it short. I focused only on implementing a rule for “Constraint”, a refactoring of the `archnemesis-sonar-rule` as configuring Jenkins to deploy it to Jenkins automatically.

So the idea was to add a constraint that enforces that a component is used together with some class or package. Or in other words, the constraint would say “if you use component X, you should also use Y”.

4.5.1 Syntax of Constraints

Here I attempted to keep with the spirit of literate programming and make code read as natural text as much as possible. The rule for a constraint was defined thusly:

```
  Constraint:
    'clients of ' component = [Component] 'must use '
       requiredClass = FullyQualifiedClassName;
```

As you can see, the syntax is made to read like a sentence and reflects the intent behind the constraint directly. A new grammar rule called `FullyQualifiedClassName` is used here. The rule is defined thusly:

```
terminal ClassName:
  '.'('A'..'Z')('a'..'z'|'A'..'Z'|'1'..'9')*
;

FullyQualifiedClassName:
  PackageName ClassName
;
```

A fully qualified class name thus consists of a package name followed by a class name. A class name corresponds to a typical Java class name, starting with a capital letter. The '.' before the actual name is there to not overshadow the `ESSAY` rule.

4.5.2 Semantics of Constraints

When the rule is processed, the `archnemesis-sonar-rule` first goes through the list of constraints. If there are constraint, it looks whether the `component` is imported into a class, and if so it looks whether the `requiredClass` is also present in the imports.

4.5.3 Evaluation

The feedback for this feature came only from one developer:

> Components in the Core cannot even use that line.
>
> Components in Eclipse can use the UISynchronizer. If there are really special cases they should just C&P this method into the component.
>
> Even the jface stuff is dangerous and this is the part where you will hit a big wall. While you may need some stuff of those packages you open access to all "wonderful" things like dialogs etc ... Which should not be used (even in Eclipse only Components) at all.

However, I had no time to draw conclusions from this comment as time for implementation ran out.

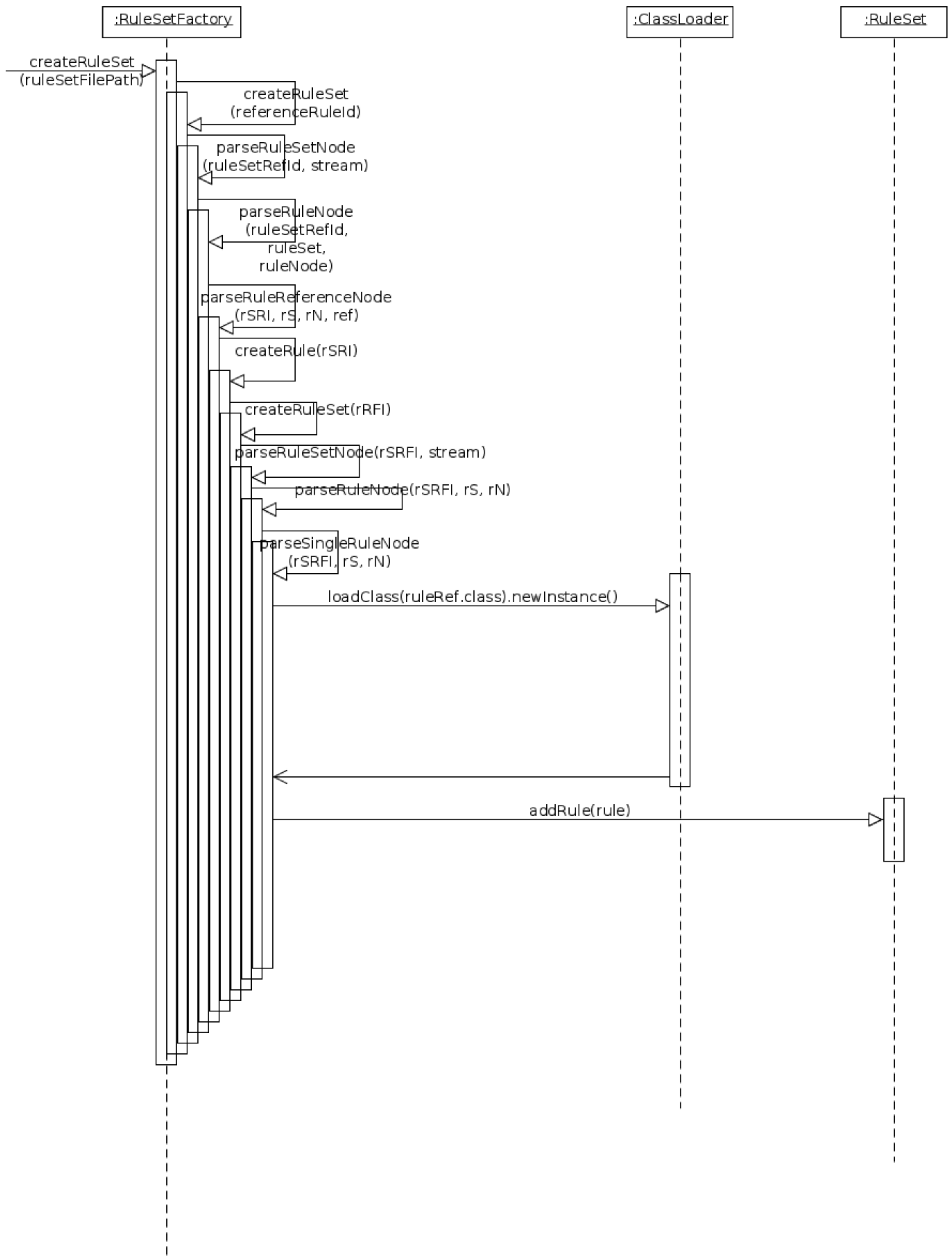
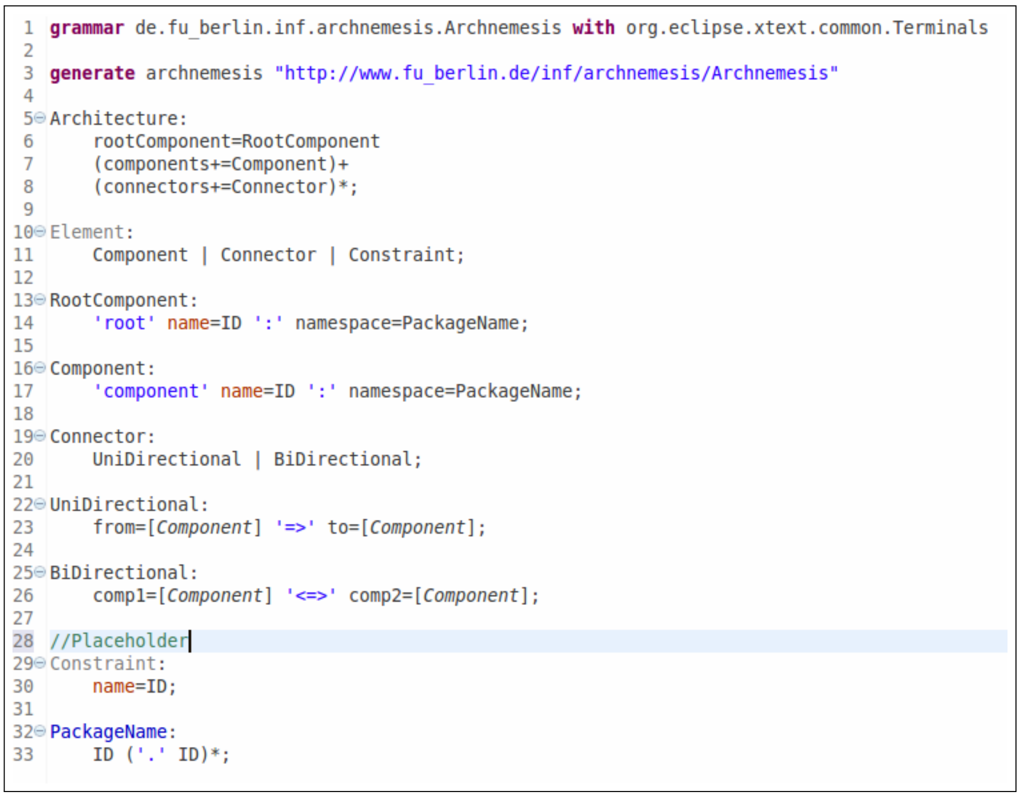
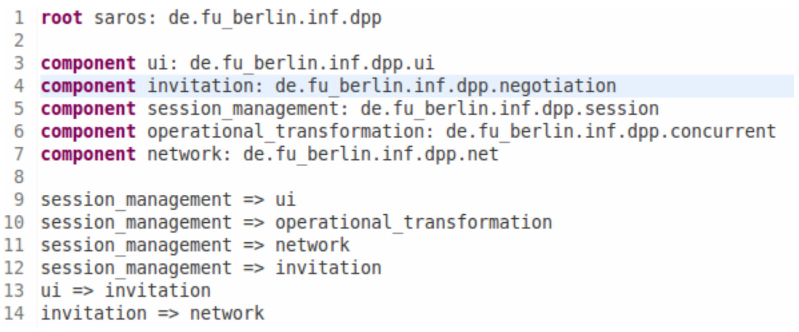

:RuleSetFactory
:ClassLoader
:RuleSet
createRuleSet (ruleSetFilePath)
createRuleSet (referenceRuleId)
parseRuleSetNode (ruleSetRefId, stream)
parseRuleNode (ruleSetRefId, ruleSet, ruleNode)
parseRuleReferenceNode (rSRI, rS, rN, ref)
createRule(rSRI)
createRuleSet(rRFI)
parseRuleSetNode(rSRFI, stream)
parseRuleNode(rSRFI, rS, rN)
parseSingleRuleNode (rSRFI, rS, rN)
loadClass(ruleRef.class).newInstance()
addRule(rule)

Figure 9: How the RuleSetFactory creates a RuleSet. Firstly each RuleReferenceId is went over. Each RuleReference contains a corresponding qualified class name of the implementing class of the rule (this qualified name is defined in the rulesets.xml). The class is then loaded through the classloader and added to the RuleSet.



```
grammar de.fu_berlin.inf.archnemesis.Archnemesis with org.eclipse.xtext.common.Terminals

generate archnemesis "http://www.fu_berlin.de/inf/archnemesis/Archnemesis"

Architecture:
    rootComponent=RootComponent
    (components+=Component)+
    (connectors+=Connector)*;

Element:
    Component | Connector | Constraint;

RootComponent:
    'root' name=ID ':' namespace=PackageName;

Component:
    'component' name=ID ':' namespace=PackageName;

Connector:
    UniDirectional | BiDirectional;

UniDirectional:
    from=[Component] '=>' to=[Component];

BiDirectional:
    comp1=[Component] '<=>' comp2=[Component];

//Placeholder
Constraint:
    name=ID;

PackageName:
    ID ('.' ID)*;
```

Figure 10: Archnemesis Grammar



```
root saros: de.fu_berlin.inf.dpp

component ui: de.fu_berlin.inf.dpp.ui
component invitation: de.fu_berlin.inf.dpp.negotiation
component session_management: de.fu_berlin.inf.dpp.session
component operational_transformation: de.fu_berlin.inf.dpp.concurrent
component network: de.fu_berlin.inf.dpp.net

session_management => ui
session_management => operational_transformation
session_management => network
session_management => invitation
ui => invitation
invitation => network
```

Figure 11: Architecture.chralx file describing the components of Saros

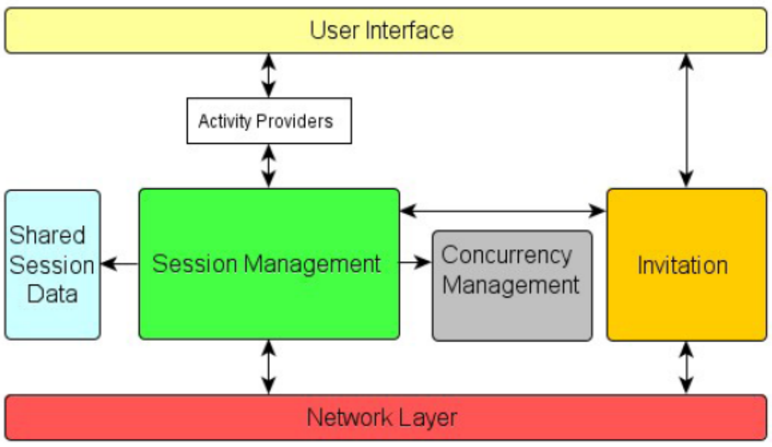


Figure 12: Architecture.chralx file describing the components of Saros

5 Conclusion and Outlook

Industriousness and conscientiousness are often antagonists, owing to the fact that industriousness wants to pluck the fruit sour from the tree while conscientiousness wants to let it hang too long, until it falls and is bruised.

FRIEDRICH NIETZSCHE, *All Too Human*

Ultimately the following results were achieved in this thesis:

- The build infrastructure of Saros was enhanced to include Sonarqube analyses.
- The results of these analyses were made to be presented to the developers in an unobtrusive fashion, as reviews in Gerrit. This way Sonarqube became integrated into the development process.
- A DSL for the description of architectures in a literate programming style called `Archnemesis` was introduced.
- A description of Saros in `Archnemesis` was made.
- A Sonarqube plugin `archnemesis-sonar-rule` was developed which can run architecture compliance checks based on a an architecture description in `Archnemesis`.
- This way the foundation for the operationalization of Saros' architecture was laid.

The `Archnemesis` Xtext project as well the Sonarqube plugin `archnemesis-sonar-rule` can be found on Saros' GitHub homepage `https://github.com/saros-project/`.

There is a series of technical problems I have not been able to solve during the duration of this thesis, due to a combination of time pressure and a lack of expertise:

- Sonarqube provides a readily comprehensible view of the code coverage in a code base. Some steps were made to achieve this, but I didn't follow through.
- Too often Sonarqube hits a read timeout during an analysis. The developer mailing list of Sonarqube has been contacted regarding this issue. The reply was to increase the timeout threshold, however this only reduced the symptoms.

- The script that transforms an issues report into a REST request to Gerrit should be integrated the Issue Report Plugin. This script would fit nicely as an output format in the plugin, and would prevent possible incompatibilities if the format of the issue reports were to change.

At the end of Chapter 6 one can see that the "Constraint" feature was not matured enough. This is indicative of what I think is the greatest shortcoming of this thesis, namely understanding Saros' architecture well enough to develop a better thought-out out DSL, which fits the needs of Saros better. However, I feel that a sound basis for further development has been laid. Moreover, this basis is completely open-source and I feel that some of the artifacts can be integrated into larger projects. For instance

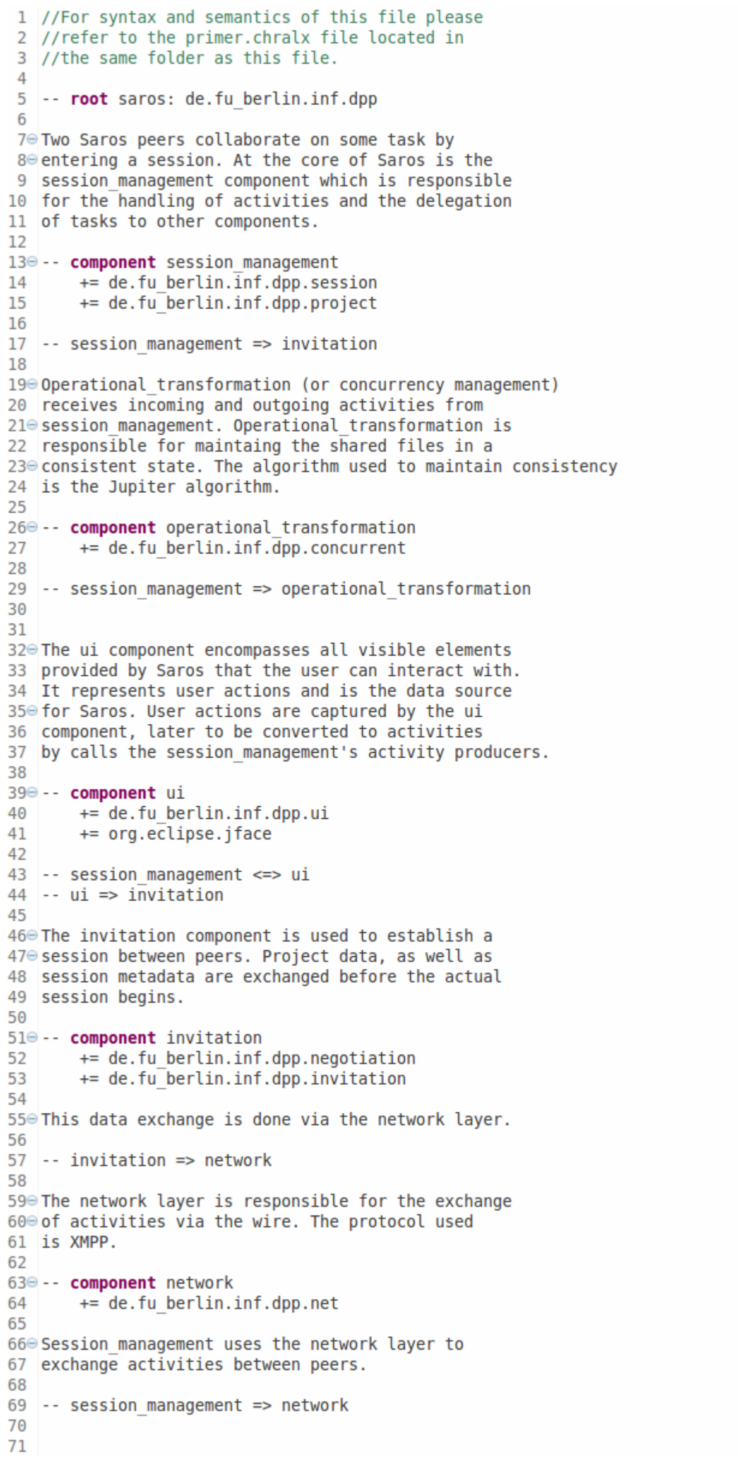
- The integration scripts, which enable the posting of issue reports can be used to integrate Sonarqube and Gerrit, where Sonarqube issues are directly translated into a Gerrit review.

- The Sonarqube plugin together with `Archnemesis` could be presented to the Sonarqube community.

A better look can be taken at ADLs, and more specifically the way they define constraints in order to find more mechanisms for constraints. A better look can be taken at Saros, to find further key architectural aspects. One could also consider how `Archnemesis` could define a Server/Client architecture, as this is in ongoing effort in the research group[32].

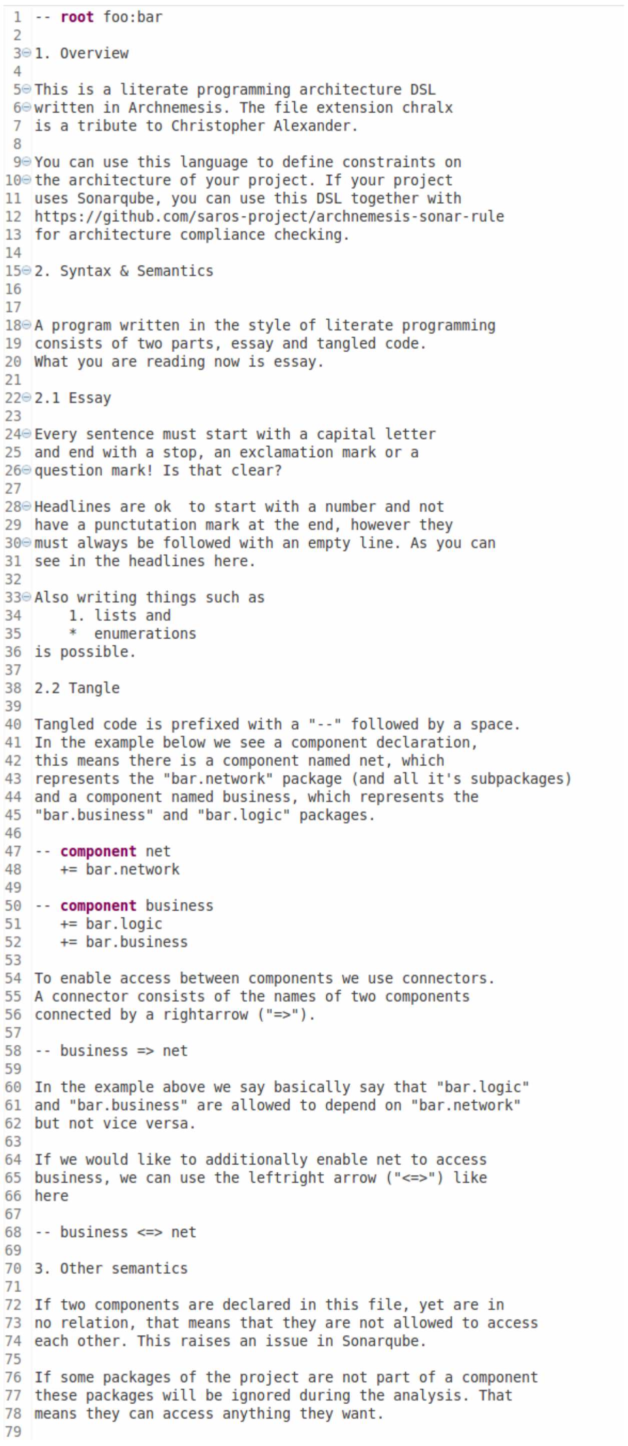
[32]This is a proposed topic for theses by the research group. See: `http://www.inf.fu-berlin.de/w/SE/ThesesDPP#Saros_45Server:_Realisierung_Nutzer_45unabh_228ngiger_Sitzungen_40B_44_M_44_D_41`

6 Appendix



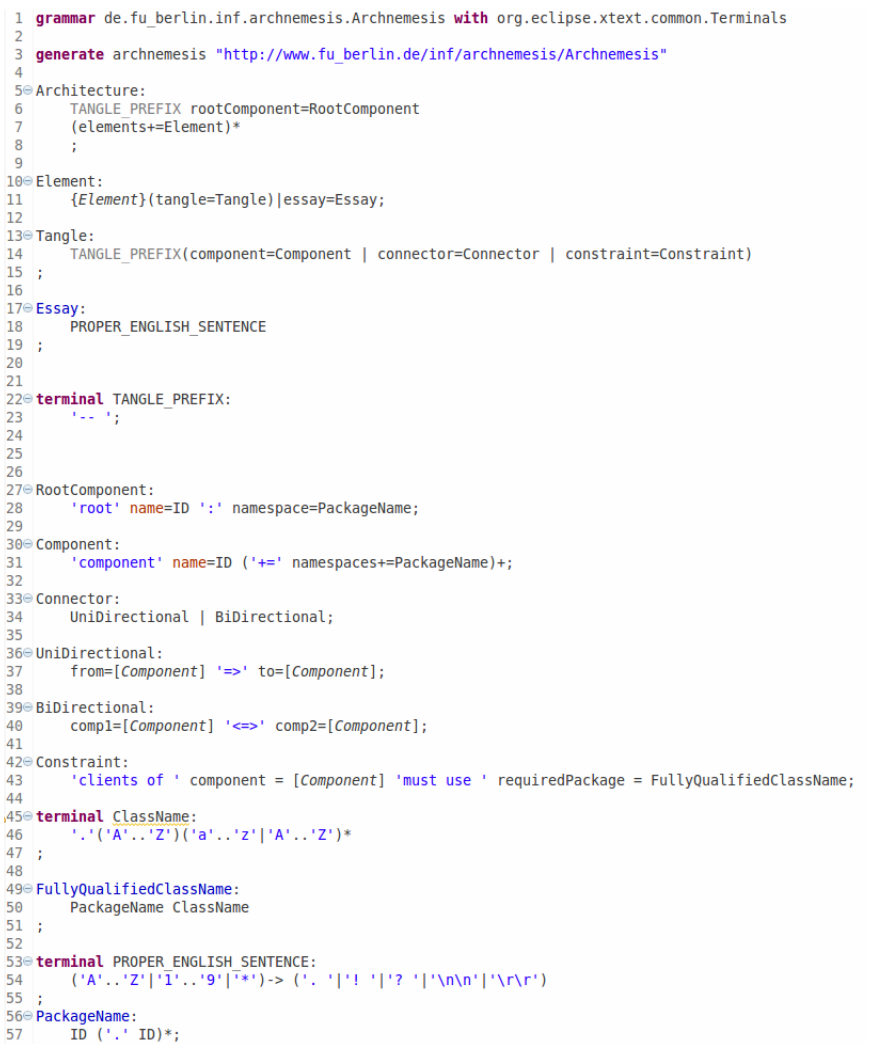
```
//For syntax and semantics of this file please
//refer to the primer.chralx file located in
//the same folder as this file.

-- root saros: de.fu_berlin.inf.dpp

Two Saros peers collaborate on some task by
entering a session. At the core of Saros is the
session_management component which is responsible
for the handling of activities and the delegation
of tasks to other components.

-- component session_management
    += de.fu_berlin.inf.dpp.session
    += de.fu_berlin.inf.dpp.project

-- session_management => invitation

Operational_transformation (or concurrency management)
receives incoming and outgoing activities from
session_management. Operational_transformation is
responsible for maintaing the shared files in a
consistent state. The algorithm used to maintain consistency
is the Jupiter algorithm.

-- component operational_transformation
    += de.fu_berlin.inf.dpp.concurrent

-- session_management => operational_transformation


The ui component encompasses all visible elements
provided by Saros that the user can interact with.
It represents user actions and is the data source
for Saros. User actions are captured by the ui
component, later to be converted to activities
by calls the session_management's activity producers.

-- component ui
    += de.fu_berlin.inf.dpp.ui
    += org.eclipse.jface

-- session_management <=> ui
-- ui => invitation

The invitation component is used to establish a
session between peers. Project data, as well as
session metadata are exchanged before the actual
session begins.

-- component invitation
    += de.fu_berlin.inf.dpp.negotiation
    += de.fu_berlin.inf.dpp.invitation

This data exchange is done via the network layer.

-- invitation => network

The network layer is responsible for the exchange
of activities via the wire. The protocol used
is XMPP.

-- component network
    += de.fu_berlin.inf.dpp.net

Session_management uses the network layer to
exchange activities between peers.

-- session_management => network


```

Figure 13: The second version of the architecture.chralx file, with literate programming and multi-package components



```
-- root foo:bar

1. Overview

This is a literate programming architecture DSL
written in Archnemesis. The file extension chralx
is a tribute to Christopher Alexander.

You can use this language to define constraints on
the architecture of your project. If your project
uses Sonarqube, you can use this DSL together with
https://github.com/saros-project/archnemesis-sonar-rule
for architecture compliance checking.

2. Syntax & Semantics


A program written in the style of literate programming
consists of two parts, essay and tangled code.
What you are reading now is essay.

2.1 Essay

Every sentence must start with a capital letter
and end with a stop, an exclamation mark or a
question mark! Is that clear?

Headlines are ok  to start with a number and not
have a punctutation mark at the end, however they
must always be followed with an empty line. As you can
see in the headlines here.

Also writing things such as
    1. lists and
    *  enumerations
is possible.

2.2 Tangle

Tangled code is prefixed with a "--" followed by a space.
In the example below we see a component declaration,
this means there is a component named net, which
represents the "bar.network" package (and all it's subpackages)
and a component named business, which represents the
"bar.business" and "bar.logic" packages.

-- component net
   += bar.network

-- component business
   += bar.logic
   += bar.business

To enable access between components we use connectors.
A connector consists of the names of two components
connected by a rightarrow ("=>").

-- business => net

In the example above we say basically say that "bar.logic"
and "bar.business" are allowed to depend on "bar.network"
but not vice versa.

If we would like to additionally enable net to access
business, we can use the leftright arrow ("<=>") like
here

-- business <=> net

3. Other semantics

If two components are declared in this file, yet are in
no relation, that means that they are not allowed to access
each other. This raises an issue in Sonarqube.

If some packages of the project are not part of a component
these packages will be ignored during the analysis. That
means they can access anything they want.

```

Figure 14: The first version of the Archnemesis primer, which explains and demonstrates the syntax and semantics of the DSL



```
grammar de.fu_berlin.inf.archnemesis.Archnemesis with org.eclipse.xtext.common.Terminals

generate archnemesis "http://www.fu_berlin.de/inf/archnemesis/Archnemesis"

Architecture:
    TANGLE_PREFIX rootComponent=RootComponent
    (elements+=Element)*
    ;

Element:
    {Element}(tangle=Tangle)|essay=Essay;

Tangle:
    TANGLE_PREFIX(component=Component | connector=Connector | constraint=Constraint)
;

Essay:
    PROPER_ENGLISH_SENTENCE
;


terminal TANGLE_PREFIX:
    '-- ';



RootComponent:
    'root' name=ID ':' namespace=PackageName;

Component:
    'component' name=ID ('+=' namespaces+=PackageName)+;

Connector:
    UniDirectional | BiDirectional;

UniDirectional:
    from=[Component] '=>' to=[Component];

BiDirectional:
    comp1=[Component] '<=>' comp2=[Component];

Constraint:
    'clients of ' component = [Component] 'must use ' requiredPackage = FullyQualifiedClassName;

terminal ClassName:
    '.'('A'..'Z')('a'..'z'|'A'..'Z')*
;

FullyQualifiedClassName:
    PackageName ClassName
;

terminal PROPER_ENGLISH_SENTENCE:
    ('A'..'Z'|'1'..'9'|'*')-> ('. '|'! '|'? '|'\n\n'|'\r\r')
;
PackageName:
    ID ('.' ID)*;
```

Figure 15: The final version of the Archnemesis grammar, features include component and connector definitions. Literate programming is supported with the ESSAY element. A simple constraint mechanism is also available.

References

[All97] Robert J. Allen. *A Formal Approach to Software Architecture*. PhD thesis, Carnegie Mellon University, 1997.

[Aug13] Anne Augustin. Integration von architekturmodellen in einen agilen softwareentwicklungsprozess. Master's thesis, FU Berlin, 2013.

[BBvB$^+$] Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, and Dave Thomas. The agile manifesto. `http://agilemanifesto.org/`.

[Bel11] Wjatscheslaw Belousow. Verbesserung der architektur der dpp-software saros durch einfÃ$\frac{1}{4}$hrung einer dokumentierten modulsicht. Master's thesis, FU Berlin, 2011.

[BW14] Georg Buchgeher and Rainer Weinreich. Continuous software architecture analysis. In *Agile Software Architecture: Aligning Agile Processes and Software Architecture*. Morgan Kaufmann, 2014.

[CHCM14] Jean Cleland-Huang, Adam Czauderna, and Mehdi Mirakhorli. Driving architectural design and preservation from a persona perspective in agile projects. In *Agile Software Architecture: Aligning Agile Processes and Software Architecture*. Morgan Kaufmann, 2014.

[Fow] Martin Fowler. Is design dead? `http://martinfowler.com/articles/designDead.html`.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[JOC14] Trygve Reenskaug James O. Coplien. The dci paradigm: Taking object orientation into the architecture world. In Ivan Mistrik Muhammad Ali Babar, Alan W.Brown, editor, *Agile Software Architecture: Aligning Agile Processes and Software Architecture*, chapter 2, pages 25–59. Morgan Kaufmann, 2014.

[Kru95] Philippe Kruchten. Architectural blueprints: The "4+1" view model of software architecture. In *IEEE Software 12(6), pages 42-50*, 1995.

[MABM13] Alan W.Brown Muhammad Ali Babar and Ivan Mistrik, editors. *Agile Software Architecture: Aligning Agile Process and Software Architecture*. Morgan Kaufmann as an imprint of Elsevier, 2013.

[MT00] N. Medvidovic and R.N. Taylor. A framework for classifying and comparing architecture descriptions languages. *Software Engineering, IEEE Transactions on*, 26:70–93, 2000.

[Sch13] Patrick Schlott. Analyse und verbesserung der architektur eines nebenl Ìaufigen und verteilten softwaresystems. Master's thesis, FU Berlin, 2013.

[Sta14] Michael Stal. Refactoring software architectures. In *Agile Software Architecture: Aligning Agile Processes and Software Architecture*. Morgan Kaufmann, 2014.

[vdVB14] van der Ven and Bosch. Architecture decisions: Who, how and when? In *Agile Software Architecture: Aligning Agile Processes and Software Architecture*. Morgan Kaufmann, 2014.

[Ves93] S. Vestal. A cursory overview and comparison of four architecture description languages. Technical report, Honeywell Technology Center, 1993.