Master's thesis at the Human Machine Interaction and Software Technology Research Group

User-Centered Development of a JavaScript and HTML-based GUI for Saros

Bastian Sieker
Student ID: 6505060
bsieker@mail.uni-paderborn.de

First Reviewer: Prof. Dr. Lutz Prechelt (Freie Universität Berlin)
Takustraße 9, 14195 Berlin
Second Reviewer: Prof. Dr. Gerd Szwillus
Supervisor: Franz Zieris (Freie Universität Berlin)

Paderborn, Septemeber 1, 2015

Bastian Sieker

Abstract

The GUI of Saros, a plugin for Eclipse and IntelliJ for distributed collaborative programming, is ported to HTML and JavaScript to enable IDE-independent development of Saros. In this thesis, a Saros GUI based on the named technologies is developed following UCD principles. During the UCD process 19 usability problems were identified and 7 of them fixed.

Several JavaScript frameworks were evaluated based on Saros's specific requirements, AmpersandJS was identified as the most promising solution, in the end. I defined an architecture for embedding the HTML frontend into Saros based on the requirements of the Saros application and development team. Due to the high fluctuation in the Saros team, the development process is supported by tools to ease the development, especially for developers without experience in the area of web development. The result of the implementation is a solid groundwork for future developers to work on the Saros GUI and to implement missing features to be able to replace the old Saros GUI in the future.

Bastian Sieker

Affirmation of independent work

I hereby declare that I wrote this thesis myself without sources other than those indicated herein. All parts taken from published and unpublished scripts are indicated as such.

Paderborn, Septemeber 1, 2015

Bastian Sieker

Contents

1 Introduction

1.1 Saros

Saros is an open source software to enable distributed collaborative software development and is available as a plugin for the integrated development environments (IDE) Eclipse[1] and IntelliJ[2]. Saros started as a research project in the software engineering research group at Freie Universität Berlin, went open source and is further developed by the community and various thesis workers at the university.

1.2 Motivation

Currently, Saros is developed for Eclipse and IntelliJ. Each of these uses a different graphical toolkit to build the Graphical User Interface (GUI). To ease the development process of the GUI and reduce redundancies, an IDE-independent implementation was targeted. Furthermore, an IDE-independent implementation supports the development of Saros for further IDEs[3].
Cikryt[Cik15] evaluated an HTML and JavaScript-based approach with promising results and lay the groundwork for implementing a GUI based on the named technologies. However, the prototype was built to evaluate the possibility of building the GUI with HTML and JavaScript, only. Thus, a fully-functional implementation is still missing.

1.3 Goals

The main goal of this thesis is to build a new Saros GUI based on JavaScript and HTML. During development, special emphasis is on the usability of the GUI. Therefore, via iterative usability tests, early feedback will be gathered from users. To maximise the usability of the GUI in the course of this thesis, an iterative and user-centered design process must be evaluated and applied.

A second goal is to keep the entry barrier for future developers as low as possible. This is due to the high fluctuation in the development team around Saros. I will evaluate the current JavaScript framework landscape to find a suiting framework supporting this goal. Furthermore, tooling for automation of common tasks is investigated. The application and the resulting development workflow should be documented in detail, to enable even de-

[1] http://www.eclipse.org/

[2] https://www.jetbrains.com/idea/ The plugin for IntelliJ is not released, yet.

[3] http://www.inf.fu-berlin.de/w/SE/ThesesDPP#PortierungIDEs (retrieved August 25th, 2015)

velopers without experience in the field of web development to develop the application in the future.

1.4 Terminology

I abbreviate user interface (UI), graphical user interface (GUI), and integrated development environment (IDE). I will use *Saros GUI* to address the GUI developed in the course of this thesis. Whenever I have to refer to the existing SWT[4] implementation, I will explicitly call it the *old Saros GUI*.

1.5 Structure

This thesis is about implementing a new Saros GUI, based on HTML and JavaScript, following user-centered design principles. At first, the relevant literature on Saros is summarised (Section 2) and fundamental concepts of the fields of usability engineering and user-centered design are introduced (Section 3.1). Based on that, objectives and methodologies of the user-centered design process are elaborated (Section 3.2 to 3.5).
Next, essential technologies and requirements are introduced (Section 4.1 to Section 4.3), promising JavaScript frameworks, as a basis for the Saros GUI, are evaluated (Section 4.4), then additional tooling is presented (Section 4.5). A detailed look at the implementation, its challenges (Section 5.1 to 5.3) and the tooling (Section 5.4) is taken, the build integration is discussed shortly (Section 5.5) and accompanying refactorings (Section 5.6) as well as the results of the implementation are presented (Section 5.7).
Finally, the concrete process of applying the user-centered design process is explained and its results are summarised (Section 6) before a conclusion of this thesis is given (Section 7).

[4]Standard Widget Toolkit, a graphical widget toolkit for Java.

2 Related Work on Saros

2.1 Regarding Usability

There are various works regarding the evaluation and improvement of the usability of Saros. For example, M. Spiering utilised an iterative UCD process[5] to identify and solve multiple usability problems [Spi12]. Her work elaborates a mental model of end-users of Saros. Further, she compares this model to the implementation of Saros and, based on the deviation between both, identifies existing usability problems. Furthermore, Spiering collected the results of multiple preceding theses regarding the usability in Saros (for example B. Kahlert [Kah11], A. Solovjev [Sol11] and A. Waldmann [Wal12]) and provides a catalogue of usability problems. Most of the collected problems have their origin in fundamental concepts of Saros. For example, unexperienced users often have problems understanding the host-concept of Saros and therefore are confused by the special role of the host in a session. Furthermore, it is often not clear to the user when the follow mode is paused or stopped, users want to have a voice connection to ease communication or they want to have the same colour assigned in successive sessions. All of these problems are deeply linked with the fundamental concepts of Saros. To resolve them, existing functionality would have to be adapted or additional functionality implemented. Those problems can not simply be resolved via adaptions to the user interface.

2.2 Regarding Technology and Architecture

The first utilisation of HTML for GUI development in the context of Saros was done by D. Durmaz [Dur14]. Durmaz implemented a prototypical view based on the integration of a browser-widget[6] (a project initiated by B. Kahlert, then a researcher in the Software Engineering working group of the FU Berlin) and compared the performance to a SWT implementation and got promising results.

In the scope of his master thesis, C. Cikryt further elaborated the feasibility of an HTML-based GUI for Saros, motivated by the fact that Saros is developed for Eclipse as well as IntelliJ, which use different graphical widget toolkits [Cik15]. This led to designated implementations for each of both IDEs. Based on previously defined goals like reduction of duplicate code, maintainability and IDE-independency, Cikryt evaluated various technologies (regarding the embedding of a browser in the IDE) and implemented a working prototype. The main aspect of Cikryt's work was

[5] Spiering only iterated once but the process was designed and is probably suitable for multiple iterations.

[6] https://github.com/bkahlert/com.bkahlert.nebula/tree/master/src/com/bkahlert/nebula/widgets/browser

the browser embedding and the interface between Java and JavaScript, however, in addition, a GUI prototype was implemented with the web technologies JQuery[7], Bootstrap[8] and AngularJS[9]. As a part of this thesis, based on the evaluation of different JavaScript frameworks, the prototypical implementation was replaced with a new GUI built from scratch. More on that in chapter 4.

Based on Cikryt's work, M. Bohnstedt investigated the IDE-independet development of Saros and identified properties and potential for optimisation in the software architecture as well as the development process [Boh15]. Furthermore, he improved and extended the interface for the Saros GUI on Java-side. There is a close relation between my thesis and Bohnstedt's thesis since we both worked on the interface between the Java and the JavaScript application, in parallel. My thesis focuses the GUI implementation and the encapsulation of the interface on JavaScript-side. Due to this close relation and resulting dependencies, collaborative work was indispensable between us.

[7] https://jquery.org/
[8] http://getbootstrap.com
[9] https://angularjs.org/

3 User-Centered Design Process

Section 3.1 gives an introduction for the terms *Usability* and *User-Centered Design*. Next, Section 3.2 defines objectives of the usability evaluation in the course of this thesis. Section 3.3 introduces and evaluates relevant usability engineering methods. Finally, in Section 3.4 the intended process of evaluating the Saros GUI is presented.

3.1 Introduction

3.1.1 Usability

According to the International Standards Organization the term *Usability* describes:

> "...the extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use." [Int98]

Usability problems are defined by Karat et al. as anything that interfere with a user's ability to efficiently and effectively complete tasks [KCF92]. In the course of solving usability problems the field of *Usability Engineering* originated. Usability engineering provides processes and methods to achieve specific attributes which are defining the usability of a product. The work of Nielsen [Nie93] is a major contribution in this field.

3.1.2 User-Centered Design

> "User-Centered Design (UCD) is a multidisciplinary design approach based on the active involvement of users to improve the understanding of user and task requirement, and the iteration of design and evaluation." [MVSC05, p.105]

ISO 13407 [Int99] provides a definition for the *Human-Centered Design Process for Interactive Systems* which is the basis for most UCD processes. It defines four activities which are part of the iterative design process:

Context of Use Identify users (and their characteristics), the purpose and the context of use of the product.

Requirements Identify the requirements of the user for the product.

Design Produce design solutions (various stages possible, for example paper prototyping or the actual software implementation).

Evaluation Evaluate designs against requirements to record the progress and to decide whether a new iteration is necessary (for example via user tests, interviews etc.).

ISO 13407 defines the workflow of the design process without specifying any practical methods.

3.2 Objective

In the following we will define the objectives of the UCD process. Nielsen introduces the terms *formative* and *summative evaluation* [Nie93, p.170]. *Formative evaluation* is about identifying good and bad parts of an interface and how they influence the usability. *Summative evaluation* is about measuring the quality of an interface, for example to compare it to other products. Furthermore, Nielsen says about *formative evaluation*:

> "Formative evaluation is done in order to help improve the interface as part of an iterative design process." [Nie93, p.170]

In this thesis, *formative evaluation* is focused. Since the UI is build from scratch and there are dependencies to other theses, it is not clear whether in each test iteration the same range of features is implemented. Thus, it will be difficult to compare test results between iterations. Since the goal is to simply minimise usability problems instead of measuring usability, this should not be a problem.

3.3 Methods

In the following, different usability engineering methods are introduced and evaluated with their cost-value ratio in mind. The costs of the applied methods should be reasonable with the limited time available for this thesis in mind. Another important factor is the number of test users necessary to get reliable results for a given test method, since it is difficult to gather a big test user base.

3.3.1 Thinking Aloud User Tests

User tests with real users is the most fundamental usability method and is in some sense irreplaceable, since it provides direct information about how people use computers and what their exact problems are with the concrete interface being tested [Nie93, p. 165]. Even with only a small group of test users you can achieve very good results, however the effort is quite high.

The *Thinking Aloud* methodology is a user test where the user is continuously thinking out loud while using the system [Nie93, p. 195]. This gives insights about why a user performs a specific action and what the user

expects of it, thus it helps understanding the user. However, it is important to have a critical view on the user's feedback. Often they develop their own biased theories about why they acted like they did. Therefore, it is very helpful to record the test session to have a look at the situations in question, later on. Another challenge is to make the user thinking out loud without forcing him to much, since this is unnatural behaviour to most people.

Independently of whether performing a user test with thinking aloud or not, there are a number of parameters to weigh up during the planning period. In the following I will discuss some of those parameters[10] with respect to the application to test, the Saros GUI.

Who are the test users? Are they experienced with the software?
The test users should have a background in the field of software development since that is the target group of Saros. It is not mandatory for a test user to have experience with Saros (novice users), however users with experience (expert users) are interesting because they know the old GUI of Saros and may identify consistency issues, for example. This is important because despite building the new GUI from scratch, one goal is to keep reasonable conventions and workflows to ease the transition from the old to the new implementation for expert users (however, not at the expense of the general usability). Hence, a mix of novice and expert users would be optimal.

Furthermore, there is a distinction between *within-subject-testing*, where one test user is participating in multiple iterations, hence is testing different versions of the GUI, and *between-subject-testing*, where one test user participates in one iteration, only. However, as Nielsen states, this distinction is mainly important in summative evaluations. However, *Within-subject-testing* may lead to interesting results when the user tests a feature for the second time and a previously reported usability problem is fixed. That gives additional feedback to the developer whether the problem is fixed, sufficiently.

What tasks will the users be asked to perform?
The tasks will depend on the number of features implemented at the time of the test[11]. I will cover all implemented, essential features in each test session since the range of those features is manageable.

How many test users are needed?

[10] Only a subset of parameters defined by Nielsen is discussed, for the full list of parameters see [Nie93, p. 170].

[11] As mentioned before, there are dependencies to other theses and it is difficult to plan when a certain feature is implemented and ready to test.

According to Nielsen, the optimal number of test users, with respect to the ratio of benefits to cost, is five [Nie00]. Since multiple iterations will be done and it is difficult to find volunteers, the number of test users should be between three and five per iteration.

What data is going to be collected, and how will it be analysed?
Screen and audio recordings (cameraless videotaping) can be very helpful to analyse the test session in more detail, retrospectively. During the test, the experimenter can take notes whenever the test user experiences problems, however there is not enough time to completely analyse the situation. This can be done afterwards with the recordings.

Who is the experimenter?
For convenience and limited resources (I don't have to look for an additional experimenter) I will take the role of the experimenter. It is no problem to do so generally, however, there are some risks like a possible lack of objectivity and the tendency to explain problems away [Nie93, p.180]. Thus, I try to strictly follow the "shut-up" rule: Intervention is only allowed in situations where the user got stuck, completely.

3.3.2 Questionnaires and Interviews

Questionnaires and interviews are a beneficial way of gathering additional information regarding the subjective perception of the tested system by the user. Such information is hard to measure objectively, for example whether a user is stressed or which features of the system are especially good or bad. However, as already mentioned about the thinking aloud methodology: *"Data about people's actual behaviour should have precedence over peoples claims of what they think they do."* [Nie93, p.209].

Questionnaires and interviews are very similar methods. Since interviews are more flexible and the users are in place after the user test anyway, interviews may be more efficient. One can respond more flexible to the statements of the user and go into detail of specific issues. In addition, you can explain a question in more detail if the user does not understand it correctly.

3.3.3 Heuristic Evaluation

Heuristic evaluation is a method based on a set of rules to identify certain patterns and characteristics of a user interface and to asses them. Nielsen defines 10 heuristics for the design of user interfaces which are more or less universally applicable [Nie93, p.209]. Heuristic Evaluation is a good method to find usability problems early to be prepared for test with actual users.

Nielsen suggests to use more than one evaluator (however, one is better than none) where each is working in isolation to ensure unbiased evaluations [Nie93, p.157]. The evaluators can freely explore the UI and should try to cover all offered functionalities. It is reasonable to go through the UI multiple times. In the first run they can get a feeling for the flow of interaction and the scope of the system, in the second run they can have a more detailed look on smaller elements and whether they fit in the global system.

3.4 Intended Process

Based on the previously evaluated methods, in the following I will describe the concrete plan for the realisation of the UCD process in this thesis. All in all, the thinking aloud loud methodology is very good for collecting qualitative data with a relatively small amount of users, hence, it will be the most important methodology for testing the Saros GUI.

3.4.1 When to Start Testing

The core idea of UCD is to get early feedback from real users to be able to respond to their requirements while developing the product. However, the product, in this case the Saros GUI, should be in a stage where it is reasonable to test. For example, it is not reasonable to test a feature which is not implemented completely and has obvious flaws. In this case, a user test is waste of time because the bigger part of problems which will arise are already known. In addition, the results of those tests may be useless in the end, since the behaviour and handling of the feature may change significantly during further implementation.

Even tests of fully implemented features can be needless. For example, it may be useless to test a feature of the GUI in isolation because this feature makes no sense to the user without knowing how it will be integrated in the overall system. Therefore, when testing, the system should be in a state where the implemented features are reasonably embedded in the the system and the desired workflow.

3.4.2 Process

The following process of evaluating the usability of the Saros GUI is pursued, roughly:

1. Heuristic Evaluation
2. Iterative UCD Process
   (a) Iteration 1
   (b) Iteration 2
   (c) Iteration 3

Before actually performing user tests, a heuristic evaluation should be performed to identify and eliminate the most evident usability problems and major flaws of the system. Afterwards, it is planned to have three iterations of user tests, each iteration with three to five users. The GUI should be tested to the extend it is implemented, at the time. In each iteration and with each test user a test session will have the following stages[12]:

1. **Preparation** Setup the test room and the software environment. Everything should be ready to start when the user arrives.
2. **Introduction** Welcome the user, give an introduction to the test session with information like what will be done, what will be tested, what will be recorded, that the user can abort the test at any time, etc. Explain the thinking aloud methodology and eventually answer questions of the user.
3. **Thinking aloud user test** Run the test according to previously defined tasks/scenarios. Make notes of interesting situations, also to have some material for discussion in the interview.
4. **Interview** The interview will be held immediately after the test session. Except some standard questions like *What did you especially like or dislike?* it will be a free interview where interesting situations from the test session are discussed in more detail, for example with questions like *What did you expect will happen after performing this particular action?*.
5. **Debriefing** Give the user insights about why the test session is helpful, what it contributes to this thesis and answer questions.

[12] Based on *"Stages of a Test"* [Nie93, p. 187]

Table 1: Notation for problem documentation

#No	Title	Fatality
Problem description		
Source		
Approach		
Discussion		

3.4.3 Test Tasks

The test tasks should be designed in a way that the fundamental functions are covered in realistic scenarios. In the case of Saros, at least the functionalities of starting and joining a session should be covered. They are essential for the functionality of Saros. In addition, they are the most complex parts of the GUI. Furthermore, the tasks should be restricted to IDE-independent functionalities since that is the area of influence of this thesis.

After each iteration, the identified usability problems should be gathered and evaluated (see Section 3.5 for details). Before the next iteration, the GUI should be reworked according to the identified problems with respect to the severity and the effort necessary to solve them.

3.5 Documentation of Results

The problems identified during the UCD process are documented on the basis of the notation introduced by Spiering [Spi12, p.59]. Table 1 shows the template for the documentation of a usability problem. The cell *Fatality* specifies the severity of the problem based on Table 2. Furthermore the source of the problem is explained (*Source*), a detailed description is given (*Description*) as well as suggestions how solve the problem (*Approach*). If I fixed the problem in the course of this thesis, the corresponding commit is referenced. Finally, the approach is discussed and, if a fix was made, it is evaluated (*Discussion*). See Table 3 for an example.
In this section the methodology of the UCD process was introduced. The procedure of applying this process is documented in Section 6. Furthermore, a notation for the documentation of identified usability problems was presented. You can find the collection of identified usability problems in Appendix A.3.

Table 2: Fatalities based on Nielsen and Mack [NLM94]

Fatality	**Description**
0 - No Usability Problem	"I don't agree that this is a usability problem at all"
1 - Cosmetic	"Cosmetic problem only: need not be fixed unless extra time is available on project"
2 - Minor	"Minor usability problem: fixing this should be given low priority"
3 - Major	"Major usability problem: important to fix, so should be given high priority"
4 - Catastrophic	"Usability catastrophe: imperative to fix this before product can be released"

Table 3: Example for notation of a documented usability problem.

#1	**Unclear state of contacts**	**2**
Only online contacts are marked explicitly with an *online* tag. When no contact is online, there is no indication about their state.		
Source: Usability Tests		
Approaches: • Provide a tag for every possible state and show it permanently		
Discussion: This problem was found in the old Saros GUI and was not fixed in the initial design of the new Saros GUI, thoroughly. Afterwards, the problem was fixed with the following commit: http://saros-build.imp.fu-berlin.de/gerrit/#/c/2791/		

4 Technology

This chapter introduces and evaluates the technologies of relevance for the Saros GUI. In Section 4.1, a brief introduction to utilised technologies is given. Afterwards, in Section 4.2, the Saros GUI application is compared with a common web application and special requirements are derived. Next, in Section 4.3, the communication interface between Java and JavaScript, mostly implemented by Cikryt, is introduced. In Section 4.4, there is a comparison and evaluation of possibly suitable JavaScript frameworks to utilise in the Saros GUI. Finally, in Section 4.5, the environment and setup for additional tooling is presented.

4.1 HTML, CSS and JavaScript

HTML and CSS

HyperText Markup Language (HTML) is a markup language (rather than a programming language) for building static documents, via describing the structure of the document. Today, it is the standard for creating web pages and therefore mostly interpreted in a browser. To define the visual appearance (*style*) of HTML elements, usually CSS is, and should be, utilised [W3C].

Since there is no way to add dynamic behaviour to a HTML document without further ado[13], HTML allows to embed JavaScript code in the document to access, modify, create and delete elements in the document.

JavaScript

JavaScript is a dynamic programming language often interpreted by a browser. In today's browsers there is a rich JavaScript API to manipulate the underlying HTML document and add dynamic behaviour. With the rise of JavaScript runtime environments like NodeJS[14], however, an increasing amount of stand-alone JavaScript applications is developed.

JavaScript is utilised more and more in web development since web pages evolved from static documents to complex Single Page Applications (SPAs). SPAs are an attempt to increase the usability experience of the user and to decrease the amount of data transferred between the server and the client.

[13] CSS is able to add some dynamic behaviour like hiding/showing elements, basic animations etc., but it is not able to manage application state or handling user actions beyond simple interactions like mouse hovers, conveniently.

[14] http://nodejs.org/

To do so, SPAs use AJAX[15] requests to receive data from the server without reloading the whole page in the browser. Therefore the HTTP-request appears to be non-blocking to the user, the application state is maintained on the client and the HTML document can be updated partially (or fully or not at all), according to the data received by the request.

4.2 Saros-specific requirements

In this section we will identify special properties of the Saros GUI in comparison to the SPA scenario described in the previous section. Furthermore, based on that properties, I will derive requirements for the Saros GUI.

When we see the Saros GUI as the *client* in the SPA scenario, and the Java application as the *server*, we can make essential differences:

Loading resources Resources like CSS and JavaScript files are fetched locally from the filesystem instead of a remote server via HTTP-requests.

Data interface There is a bidirectional communication interface between the Saros GUI and the Java application, whereas in the SPA scenario, usually only the client is able to make requests[16]. Furthermore, for this communication no HTTP-requests are used. For more details on the interface between Java and JavaScript, have a look at the next section, 4.3.

Hidden browser features The Saros GUI is running in a browser, however a lot of browser features are not accessible, for example bookmarking and navigation on URL-level (buttons to go back and forward in URL history).

In addition to technological aspects, there are special characteristics in the development team:

Team There is a high fluctuation in the Saros team since most team members are students writing their thesis. Moreover, we cannot assume developers to be experienced in web development nor with its technologies.

The above-named properties have an impact on architecture and design decisions of the application, which are derived in the following:

Size of artefacts Since CSS and JavaScript resources are available locally, performance in terms of loading times, which is usually very important in web applications, can be neglected. Tasks like minimising the size of the source code (*minifying*) are not neccessary.

[15] Asynchronous JavaScript and XML

[16] There are exceptions for that: with protocols like WebSocket it is possible for the server to communicate with the client, without the need of polling on client-side.

Framework choice Choose frameworks which are easy to use and learn even for developers with no or less experience with web technologies. The frameworks should allow high customisation to be prepared for special requirements. Furthermore, the file size of the utilised framework is not a major point for the evaluation of its feasibility for this project.

Tool support Utilise tools like automated code-style checker and unit-test frameworks to enable developers to find defects early.

Documentation Provide a good documentation of the code as well as of the development process and the utilised tools to ease the entrance for new developers.

All these points must be considered while designing and structuring the application. That process is explained in more detail in section 5. Next, the communication interface between Java and JavaScript is explained in more detail.

4.3 Communication interface between Java and JavaScript

After At first, we will have a look at the groundwork and the dependencies for the communication interface. Cikryt introduces the IDE-independent module `ui`, which is, as the `core` module of Saros, a bundle in Eclipse's OSGI[17] context [Cik15]. The `ui` module contains all IDE-independent Java classes and the HTML, CSS and JavaScript resources. Tasks of the Java part are:

1. Providing a bidirectional data interface between the `core` module and the `ui` module.
2. Delegation of method calls from `ui` module to the `core` module.
3. Holding application state information and their conversion to JSON[18] presentations (via GSON[19]).

In the following, the data transfer between the `ui` module and the JavaScript application running the GUI is introduced. When I speak about *Java* or *the Java-side* the `ui` module is referenced. When I speak about *the JavaScript-side* or the *JavaScript application*, I mean the JavaScript application running the Saros GUI.

[17] A modular service platform for Java, see http://www.osgi.org/Specifications/HomePage (retrieved 12th August 2015) for more information.

[18] JavaScript Object Notation

[19] A library for converting Java objects to JSON and vice versa, https://github.com/google/gson/.

Java-to-JavaScript Data Interface

From Java you can execute JavaScript code in a browser instance of the Saros GUI via calling `browser.run(String code)`. For example, if we want to call a JavaScript function named `showError` with one argument we can do it like shown in Listing 1.

Listing 1: Calling a JavaScript function from Java

```
String errorMsg = "Error";
browser.run("showError('" + errorMsg + "')");
```

There are two main use cases for calling JavaScript from Java. First, it is used to propagate application state to the JavaScript application. If for example a user successfully connected, the JavaScript application needs to get information to be able to reflect the change of state in the UI. This is done by creating JSON representations of the Java objects managing the state of the application and calling dedicated functions with the JSON representations as argument. This JavaScript function has to be defined and it has to know in which form the data is provided and what to do with it. The second use case is providing feedback to the JavaScript application in case of errors on the Java-side.

In theory, you can execute arbitrary JavaScript code, however we should seek a narrower and more explicit interface to be able to document it, appropriately. This abstraction was introduced by Matthias Bohnstedt [Boh15]. He introduces the class `JavaScriptAPI` which is responsible for encapsulating all possible JavaScript calls from Java. The direct usage of `browser.run(String code)` should be avoided, instead the well-documented methods from `JavaScriptAPI` should be used. Listing 2 shows the usage of `JavaScriptAPI`.

Listing 2: Calling a JavaScript function from Java with JavaScriptAPI

```
String errorMsg = "Error";
JavaScriptAPI.showError(browser, errorMsg);
```

The available methods can be documented as usual in Java and the whole interface is encapsulated in one class to provide a clean summary of the interface. In comparison, with the direct method in Listing 1, there is no way to conveniently document the behaviour of the JavaScript call on Java-side and no central place to encapsulate the interface.

After being able to call JavaScript from Java, next, we will have a look at how to call Java from JavaScript.

JavaScript-to-Java Data Interface

There are Java functions, in the following called browser functions, which can be invoked from JavaScript. The use case for calling Java from JavaScript is to forward actions the user triggered in the GUI to the Java-side and actually perform that action.

Browser functions are implemented in Java and inject a JavaScript function in the global namespace of the HTML document, thus, they are callable from JavaScript. Calling such JavaScript functions triggers the execution of the corresponding Java code. Those functions define the Saros API which is accessible from the JavaScript application. By default, browser functions are executed synchronously in the same thread as the UI and are therefore blocking. The return value of the function can be handled on Javascript-side, when necessary.

Listing 3: Synchronously executed browser function

```
new JavascriptFunction("__java_addContact") {
    @Override
    public Object function(final Object[] arguments) {
        try {
            stateFacade.addContact(new JID((String)
                arguments[0]),
                 (String) arguments[1]);
        } catch (XMPPException e) {
            //[...] handle exception
        }

        return null;
    }
}
```

The execution of the browser function in Listing 3 can be triggered from JavaScript like illustrated in Listing 4.

Listing 4: Calling a browser function from JavaScript

```
__java_addContact(jid, name);
```

It is also possible to swap the execution of browser functions in a different thread to make them asynchronous. If you want to work with the return value of asynchronous functions on JavaScript-side, this is not possible, directly. As a workaround you can pass information via the Java-to-JavaScript interface, see Listing 5, for example.

Listing 5: Asynchronously executed browser function

```
new JavascriptFunction("__java_connect") {
    @Override
    public Object function(Object[] arguments) {
        if (arguments.length > 0
            && arguments[0] != null) {

            Gson gson = new Gson();
            final Account account = gson.fromJson(
                (String) arguments[0], Account.class);
            ThreadUtils.runSafeAsync(LOG,
                new Runnable() {
                    @Override
                    public void run() {
                        stateFacade.connect(account);
                    }
                }
            );
        } else {
            JavaScriptAPI.showError(browser,
                errString);
        }
        return null;
    }
}
```

As mentioned, the browser functions are injected in the global namespace of the HTML document with the naming convention `__java_<functionName>`. However, it is a good idea to abstract the interface to make it clean and more convenient. This is done by introducing a `SarosApi` object on JavaScript-side, described in Chapter 5.

4.4 JavaScript MV*-Framework Evaluation

Since the implementation of the Saros GUI leads to a sophisticated JavaScript application, many architectural decisions must be made. Currently, there is a wide and sometimes confusing landscape of JavaScript MV*[20]-frameworks to ease structuring, development, maintanance, and testing browser-based applications. Most of these frameworks tackle the same problems, however, the underlying concepts diverge, heavily. Furthermore, in the context of Saros, we have to consider some additional requirements as described in section 4.2.

[20] Popular architectural pattern where the model and view part can be defined analog to the corresponding parts of the Model-View-Controller pattern. However, the controller is not further specified. For example, the view part may be responsible for doing the controller work of an traditional Model-View-Controller implementation.

To overview the various solutions and its features the TodoMVC project was started by Addy Osmani[21]. The project provides *"the same application implemented using MV* concepts in most of the popular JavaScript MV*-frameworks of today"*. To get a more detailed impression of how different frameworks work, I recommend to have a look at the corresponding implementation of TodoMVC. In the following, the most promising MV*-frameworks are evaluated with the previously defined special requirements of the use case Saros frontend in mind.

4.4.1 AngularJS

AngularJS[22] is a very popular[23] JavaScript framework which extends HTML with declarative markup to allow developers to build sophisticated UIs and wire up components without deep knowledge of JavaScript. AngularJS is feature-rich, with built-in solutions for tasks like two-way data-binding (the model updates the view on changes, automatically, and vice versa), form validation and much more. In addition, a module system and a mechanism for dependency injection is integrated. There is no additional template engine utilised, rather HTML itself is extended to perform templating.

Due to favouring declarative code over imperative code, the framework has a very high level of abstraction. Many common problems can therefore be tackled fast and efficient, however, when facing special requirements and extraordinary problems which require customising default behaviour, things can get complicated. For that purpose, the official AngularJS website states:

> "Angular simplifies application development by presenting a higher level of abstraction to the developer. Like any abstraction, it comes at a cost of flexibility. In other words, not every app is a good fit for Angular." [Ang]

A plus for AngularJS is the very big development community and its rich plugin and extension ecosystem.

4.4.2 EmberJS

EmberJS[24] is claiming itself being a *"framework for creating ambitious web applications"*. It abstracts a lot of the common tasks in the area of web

[21] http://todomvc.com/

[22] http://angularjs.org/

[23] Over 40.000 Stars on GitHub (https://github.com/angular/angular.js), roughly twice as much as BackboneJS (https://github.com/jashkenas/backbone/). EmberJS (https://github.com/emberjs/ember.js) has only around 14.000. Numbers as of August 2015.

[24] http://emberjs.com/

development like data management, routing, data-binding and templating. This is especially interesting for large applications which are long-running and have to handle a lot of data which may be strongly interconnected. It offers sophisticated functionality like creating application-specific HTML tags using Handlebars[25] to encapsulate custom markup and behaviour. To wire up the objects of an application, EmberJS strongly utilises naming conventions.

Due to the high level of abstraction, naming conventions and idiosyncratic patterns, the framework is hard to learn and internal processes are difficult to comprehend. To understand the underlying concepts it is not sufficient to be familiar to JavaScript, the codebase is huge and complex.

4.4.3 BackboneJS

BackboneJS[26] mainly provides skeletons for models, collections and views, a router and an event system to help structuring web applications. The code is well documented and relatively easy to understand due to its low level of abstraction. It does not try to provide built-in solutions for every common task (for example templating and data-binding), however it is designed to conveniently integrate such functionality, if required. There is a rich landscape of extensions for the tasks BackboneJS is not handling itself. Developers often can chose between different solutions for a problem, following different design philosophies.

On the one hand, due to the low level of abstraction, at the beginning of a project developers often have to write more boilerplate code than with frameworks like AngularJS and EmberJS. On the other hand, due to its flexibility, BackboneJS is especially interesting for custom or partially unknown requirements. There are a number of frameworks based on BackboneJS, each adding functionality to provide solutions for additional tasks.

4.4.4 AmpersandJS

AmpersandJS[27] is a highly modularised framework based on many ideas of BackboneJS. It consists of independent modules hosted on GitHub[28] which are accessible via NPM[29], the NodeJS package manager. It extends BackboneJS by features like declarative data-binding, derived properties, nested views and more. AmpersandJS shares a lot of conventions with BackboneJS, which offers the possibility to use BackboneJS components,

[25] Template Engine, http://handlebarsjs.com/.
[26] http://backbonejs.org/
[27] http://ampersandjs.com/
[28] https://github.com/AmpersandJS
[29] https://www.npmjs.com/

Table 4: Properties of evaluated frameworks

	Abstraction	Built-in features	Adaptibility / Flexibility	Learnability	Community
Angular	high	++	-	-	++
Ember	high	++	-	- -	+
Backbone	low	-	++	++	++
Ampersand	medium	+	+	+	-

like views, in your AmpersandJS application. A lot of ideas and concepts of AmpersandJS are derived from Human JavaScript[30], written by the main contributor, Henrik Joreteg.

A drawback of AmpersandJS is a relatively (at least in comparison to the three big players mentioned before) small community, however, the code is actively maintained and developed. The impact of this problem is decreased by the interoperability with BackboneJS components. Like BackboneJS, the codebase is small, well documented and readable, the level of abstraction is relatively low.

4.4.5 Decision making

As elaborated in Section 4.2, we have to keep the special requirements of the Saros project in mind, when making a decision about a framework for the Saros GUI JavaScript application. Making sure that the maintenance and further development is convenient even for developers without a web development background is crucial. Table 4 summarises the most important properties of the introduced frameworks.

AngularJS and EmberJS are fully equipped frameworks, offering a rich palette of features and powerful functionality to the developer. EmberJS does that at the cost of being intuitively usable. Developers have to deal with a lot of conventions and uncommon patterns which lead to a decrease of learnability. AngularJS even originated by the idea to enable people without a programming background to write powerful user interfaces and web applications. However, there are limitations when facing non-standard problems which can not be solved with the built-in features. The effort to handle such special cases is difficult to estimate. Furthermore, a deep knowledge of the framework is necessary.

In comparison to the previous two, BackboneJS and AmpersandJS are more flexible and easier to understand and to learn due to the lower level

[30] http://humanjavascript.com/

of abstraction. The built-in components are exchangeable and conveniently adaptable and extendable. The downside is more boilerplate code at the beginning of a project. However, I think a software engineer without a web development background will have a less cumbersome start into a JavaScript project based on a framework that provides predictable behaviour and understandable source code, rather than hiding complex functionality behind layers of high abstraction.

The nice thing about AmpersandJS is that it provides some enhancements and some more functionality than BackboneJS, without loosing its strengths. The small community may be a downside, but due to the interoperability with BackboneJS components it is future-safe, in my opinion. Therefore, AmpersandJS offers the best compromise and is the framework of choice for the Saros GUI JavaScript application.

4.5 JavaScript Tooling

Since JavaScript applications are getting more and more complex, there is an increasing need for supporting the development process with tools like it is nowadays common in other software environments like Java. There is a wide range of tasks that can be automated potentially, which is especially important to lower the entry hurdle for new developers. Such tasks are for example running unit tests, building an application bundle and code auto-formatting[31].

After briefly introducing NodeJS, which is the runtime environment for most of the tools regarding JavaScript development, I will give an example of how to use this environment in a JavaScript project like the Saros GUI.

NodeJS and NPM

NodeJS[32] is a JavaScript runtime environment to enable the development of JavaScript applications running outside of a browser. NodeJS is shipped with its package manager NPM[33] (Node Package Manager), which is also managing a lot of packages independent of NodeJS, like for example the modules of AmpersandJS.

Utilising NPM

The nice thing about NPM is that you can manage the JavaScript dependencies of your application and the dependencies regarding utilised tools,

[31] For a more extensive list, see http://rmurphey.com/blog/2012/04/12/a-baseline-for-front-end-developers/

[32] https://nodejs.org/

[33] https://www.npmjs.com/

for example for building the application, in one central place and with one package manager. As a consequence, a new developer can get all necessary source code and command line tools for building and testing the application with one invocation of `npm install`, for example.

The entry point of every project based on NPM packages is its `package.json`, where dependencies are defined. There is a distinction between dependencies regarding the application itself (`dependencies`), and dependencies for tools supporting the development of the application (`devDependencies`). Furthermore, there is the possibility to define `scripts` which can be run via `npm run <scriptName>` to ease the use of the utilised tools[34].

See Section 5.4 for detailed information about which NPM packages are utilised in the Saros GUI project, and how.

[34]Have a look at https://docs.npmjs.com/files/package.json for a comprehensive documentation.

5 Implementation

This chapter describes implementation details of the new Saros GUI based on HTML, CSS and JavaScript. In Section 5.1, adaptions of the groundwork by Cikryt are described. Afterwards, in Section 5.2, structure and implementation details of the new Saros GUI are explained, special challenges are presented (Section 5.3), followed by the environment for tool support (Section 5.4).
Next, the build integration is discussed shortly (Section 5.5) and accompanying refactorings are enumerated (Section 5.6). Finally, in Section 5.7 the results of the implementation are summarised.

5.1 UI modules

As described in Chapter 4, Cikryt introduced an IDE-independent, dedicated module `ui` in Eclipse's OSGi context which provided functionality for the communication interface between Java and JavaScript as well as JavaScript, CSS and HTML resources. For the implementation in the course of this thesis, I introduced a new module `ui-frontend`, in which the JavaScript, CSS and HTML resources are outsourced. The new module `ui-frontend` is a fragment of the module `ui`, which means that its content, the frontend resources, is made available in the `ui` module. See Figure 1 for a visualisation of the dependencies between individual Saros modules. Ac
There are two reasons for this decision. First, the `ui` module is now a pure Java module, the `ui-frontend` module contains frontend technologies only and may utilise its own, independent build process (see Section 5.5 for more details about the build process). The explicit separation emphasises that fact to make developers aware of it. Second, the application is basically completely independent of all other Saros modules, which means that it is possible to debug, test (manual testing as well as automated unit testing) and run the application stand-alone in a browser. The only requirement is, that the Java-to-JavaScript interface must be mocked, for example with hardcoded JSON data. Thus, it is possible to develop the application without requiring other Saros modules.

5.2 Saros GUI JavaScript application

In the following, structure and individual components of the Saros GUI JavaScript application are introduced.

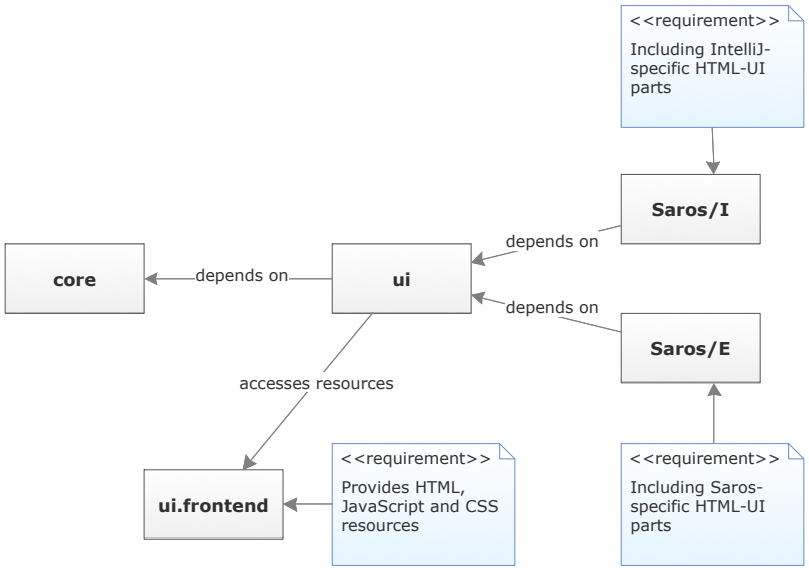

Figure 1: Dependencies between individual Saros modules, based on [Cik15, p.39]

5.2.1 CommonJS Modules

NodeJS, introduced in Section 4.5, provides a module system[35] based on the CommonJS specification[36]. It enables developers to write, share and consume modules which can be loaded via `require()`. Since the JavaScript language specifications prior to ECMAScript 2015[37] lack a module system, there are tools like Browserify (see Section 5.4 for more details), which understands `require()`[38] and can bundle browser-based applications utilising CommonJS modules.

As AmpersandJS is provided by CommonJS modules, the Saros GUI JavaScript application is structured in that way, as well. For an example, have a look at Section 5.2.3, introducing the SarosApi module.

Since AmpersandJS uses CommonJS modules, it was a nearby decision to use them for the Saros GUI, as well. It encourages the developer to modu-

[35] https://nodejs.org/docs/latest/api/modules.html
[36] http://www.commonjs.org/specs/modules/1.0/
[37] http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf
[38] JavaScript function which is called with a String describing the relative path to the required module.

larise the code, avoids the pollution of the global namespace and simplifies tasks like dependency management and unit testing.

5.2.2 Project Structure

To make it easy for developers to find the source code responsible for a specific task, the JavaScript application in `html/js` is structured in multiple directories, each holding modules of a specific type. The convention is that every JavaScript file in the project holds exactly one CommonJS module. Exceptions to that are the files in `html/js/vendor` and the entrance point of the application, `html/js/app.js`. The directory `html/js/vendor` contains all third-party code which is not available via NPM. Thus, we are not able to make assumptions of whether such code is CommonJS-compatible. For example, the Bootstrap extension for providing a context menu is placed in that directory (see Section 5.2.5 for more information about Bootstrap and other external dependencies). Since `html/js/app.js` is responsible for kicking off the application (it serves as an entry file for Browserify, see Section 5.4), it is not necessary to provide it as a module.

Models and collections (data representations) are placed in `html/js/-models`. Views, responsible for managing a specific part of the GUI, for example the contact list, are placed in `html/js/views`. Usually, they hold one or multiple models/collections and have to synchronise the data between models and their representation in the DOM. In addition, they hold a template which describes the concrete DOM representation. Those templates are written in JADE (see Section 5.2.4) and are present in `html/templates`. Furthermore, in `html/js/pages` views are placed which represent a closed entity in the Saros GUI not contained by any other page or view. For every HTML file in `html` there is exactly one page. In Figure 2, the relations between the named components of the application are visualised.

In `html/css` one can find the internal CSS definitions (`html/css/saros.css`) as well as external. Unit tests are specified in `html/test`. The application bundle generated by the build process (see Section 5.4) is placed in `html/bundle`. All files a developers wants to include into the build of the `ui-frontend` OSGi module, which are the files finally made available to the `ui` OSGi module, must be copied to `html/dist`.

5.2.3 The SarosApi Module

The interface between Java and JavaScript on JavaScript-side is completely encapsulated in the `SarosApi` JavaScript object, afterwards called SarosApi,

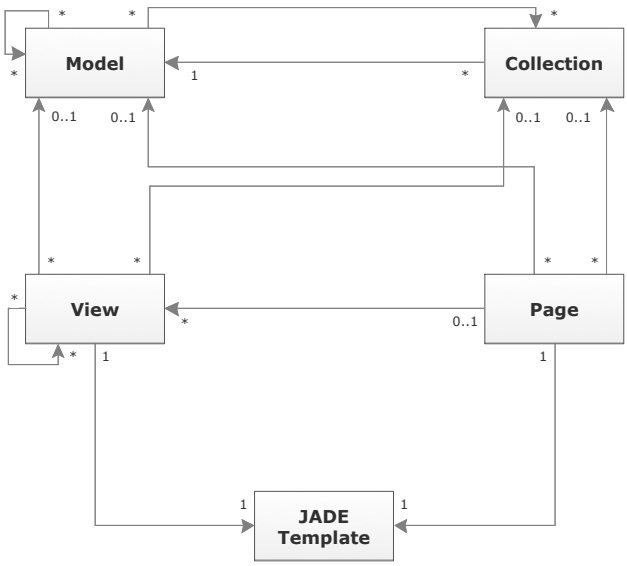

Figure 2: Relations between individual components.

which lives in `html/js/saros-api.js`. It is a usual JavaScript object literal with additional functionality to serve as an event bus and is provided to the application as a CommonJS module.

For the JavaScript to Java interface, the SarosAPI takes a role as a facade via hiding the `__java_<functionName>` functions provided by Java behind convenient functions. That way for example, instead of calling `__java_connect(account)` one can require the SarosApi and call the corresponding function like in Listing 6.

Listing 6: Requiring and using the SarosApi object

```
var SarosApi = require('saros-api');
SarosApi.connect(account);
```

In the other direction, the Java to JavaScript interface, the SarosApi serves as an event bus. Every JavaScript call made from Java should be done via triggering an event and passing data via `trigger()`. Therefore, the SarosApi object must be made available in the document, globally[39]. This happens in `html/js/app.js`, see Listing 7.

[39]Thus, in theory it would not be necessary to require SarosApi in the other modules of the application. However, requiring it nevertheless does no harm and enforces consistency.

Listing 7: Make the SarosApi object available globally

```
var SarosApi =
    window.SarosApi = require('./saros-api');
```

The JavaScript application can listen and respond to such events via `on()`, see Listing 8, for example. This messaging pattern is called *Publish/Subscribe*[40] and enables the decoupling of modules in the application.

Listing 8: Listen to events on the SarosApi object

```
var SarosApi = require('saros-api');
SarosApi.on('eventName', function(data) {
    console.log(eventName, data);
});
```

Figure 3 visualises the interface between the `ui` module and the Saros GUI JavaScript application, schematically. The SarosApi serves as a middleware for the communication between the Java and JavaScript part.

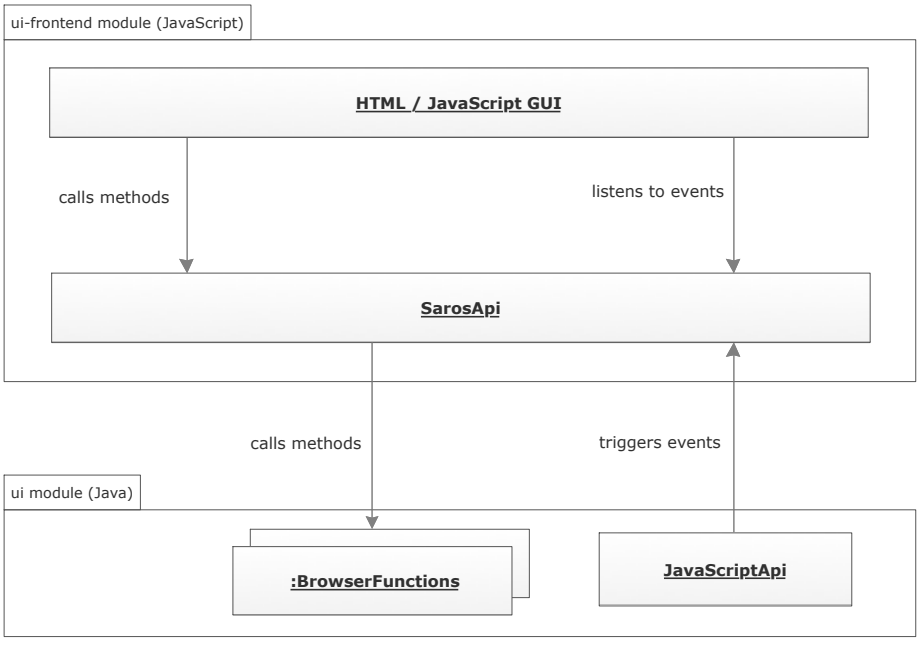

Figure 3: Architecture of the Saros GUI

The use of a module encapsulating the communication between Java and JavaScript has several advantages. First, it abstracts the cumbersome `__java_<functionName>` function calls on JavaScript-side. Second, from

[40] https://en.wikipedia.org/wiki/Publish%E2%80%93subscribe_pattern

Java-side you can conveniently trigger events with data instead of calling functions (which would have to be available, globally) or even running code directly via `browser.run(String code)`. Third, there is one central place on JavaScript-side to lookup the functions and events provided in the application.

5.2.4 HTML Templating with JADE

JADE[41] is a template engine for NodeJS producing HTML. It can be precompiled to JavaScript functions and this way used for applications running in the browser. In a complex application you want to avoid hard-coded strings in the UI to ease adopting and maintaining the application. Furthermore, in an application with the need to dynamically manipulate the DOM, it is very hard to manage the state of the DOM, manually.

Advantages of JADE are the slim representation of HTML code and the fact that invalid HTML is identified during compilation and errors are thrown, accordingly. Usually, invalid HTML is difficult to spot and resulting errors difficult to debug.

5.2.5 Additional dependencies

In the following, external libraries regarding CSS and JavaScript (in addition to AmpersandJS and jQuery) are introduced.

Bootstrap Bootstrap is a CSS and JavaScript framework which provides a lot of common UI components like buttons, tooltips, dialogs and more[42] in a appealingly and uniformly look and feel. The components are easy to integrate (mostly via using CSS classes and HTML data-attributes[43]) and theme-able[44]. Bootstrap components are utilised in nearly every component of the Saros GUI. Furthermore, an Bootstrap extension is utilised to show context menus[45].

jsTree jsTree[46] is a jQuery plugin that provides the interactive visualisation of tree structures, which are needed in the wizards for the session negotiation, for example.

[41] http://jade-lang.com/
[42] http://getbootstrap.com/components/
[43] http://www.w3schools.com/tags/att_global_data.asp
[44] http://themes.getbootstrap.com/
[45] https://github.com/sydcanem/bootstrap-contextmenu
[46] https://www.jstree.com/

Backbone.Events To be able to use the `SarosApi` as an event bus (see Section 5.2.3), a standalone version of Backbone.Events[47] is used[48].

5.3 Challenges

Due to the special environment the Saros GUI is running in (see Section 4.2), there evolved different problems and challenges in the course of this thesis, which will be presented in the following.

Debugging and Testing

As explained in an mailing list entry[49] by Björn Kahlert, there are several mechanisms provided by his browser-widget to support debugging the JavaScript application in the IDE. For example, it is possible to catch exceptions thrown by JavaScript using a registered JavaScriptExceptionListener. Calls to JavaScript's `console.log()` and `console.error()` are forwarded to the IDE console, so there is no need for using `alert()`, for example. In addition, he provided a method to open the document of the currently focused browser view (in its current state) in the system's default browser. There you can use tools provided by the browser to debug the application, conveniently. However, this functionality was removed in the improved browser-widget by C. Cikryt [50] since it is build on IDE-dependent functionality. Even with such a mechanism the debugging would be cumbersome considering alone the time until the plugin instance is running. In comparison, to build the JavaScript application and open the corresponding HTML file in a browser, stand-alone, is less costly. Therefore, the application was built with the goal in mind to be able to run it stand-alone. To do so, we must be able to provide some data to the application to test certain functionalities. If we want to test the behaviour or the visualisation of the account list, for example, we need to provide JSON data representing accounts. Luckily, it is quite easy to do so via triggering events with the corresponding data on the `SarosApi` object, see Listing 9 for an example. Such code can be placed in the `init()` function in `html/js/app.js` or run in the browser console after loading the HTML file.

Whenever we want to test parts of the application which will trigger a call of an injected function via the `SarosApi` object, we have to make sure that either the function exists (we would have to mock it in the global namespace) or we check whether it exists before calling it (both to avoid a `ReferenceError`). See Listing 10 for such an adaption in `html/js/saros-api.js`.

[47] backbonejs.org/#Events
[48] https://github.com/n1k0/backbone-events-standalone
[49] http://article.gmane.org/gmane.comp.ide.eclipse.saros.devel/1225
[50] https://github.com/ag-se/swt-browser-improved

Listing 9: Mocking accounts via the `SarosApi` object

```
SarosApi.trigger('updateAccounts', [
    {
        "username": "Alice",
        "domain": "saros-con.imp.fu-berlin.de",
        "jid": {
            "jid": "alice@saros-con.imp.fu-berlin.de"
        }
    },
    {
        "username": "Bob",
        "domain": "saros-con.imp.fu-berlin.de",
        "jid": {
            "jid": "bob@saros-con.imp.fu-berlin.de"
        }
    }
]);
```

Listing 10: `SarosApi` object adaption

```
...
connect: function(account) {

    if(typeof __java_connect !== 'undefined') {
        __java_connect(JSON.stringify(account));
    }
},
...
```

Another problem with debugging JavaScript applications based on multiple modules which are bundled in one file (like described in Section 5.4) is the mapping of a line number in the bundle to the corresponding line number in a module file to make debugging possible, conveniently. This problem is solved by source maps, which provide exactly that mapping (also, see Section 5.4).

Handling Multiple jQuery Instances

As Cikryt describes, there is an extension mechansim in the utilised browser-widget [Cik15, p.60]. For example, there is a `JQueryExtension` that injects the JavaScript code of the jQuery[51] library in the DOM, automatically. Currently, the utilised browser is indeed extended by jQuery, however, the version is outdated. Thus, a more up-to-date jQuery version must be loaded, additionally (other dependencies of the Saros GUI, for example Bootstrap, require a newer version). This up-to-date jQuery version is required in `html/js/app.js` as shown in Listing 11. The `require()` statements afterwards

[51] http://jquery.com/

load additional JavaScript code which depend on jQuery[52]. The problem is that the old, injected version is loaded after the code in Listing 11 is executed (and overrides the namespaces `$` and `jQuery`), therefore the up-to-date version is moved to a new namespace (`$$`), additionally. Thus, when referring jQuery in the application, `$$` should be used, currently. In the future, it may be reasonable to utilise a browser-widget without an extension for jQuery or updating the injected jQuery version, but this should be evaluated in more depth[53].

Listing 11: Injecting jQuery into an additional namespace

```
window.$$ = window.$ = window.jQuery =
    require('jquery');
require('bootstrap');
require('./vendor/bootstrap-contextmenu');
require('jstree');
```

Disabling Default Behaviour

In browser-based applications you want to have features like text selection and a default context menu because they are functional. However, in an application running in a browser embedded in an IDE, such behaviour is confusing and not expected by the user, as shown by the results of the usability tests (see Appendix A.3). Furthermore, it leads to inconsistencies in the UI since usually, inside other views in an IDE, those actions are not allowed. Therefore, text selection and the default context menu are disabled in the Saros GUI. Disabling the text selection is done by CSS, as shown in Listing 12. Disabling the default context menu is done without stopping the propagation of the event since it is needed in other places of the application. The `contextmenu` event is handled in `html/js/app.js`, see Listing 13.

Listing 12: Disabling text selection via CSS

```
body {
    [...]
    /* avoid text selection */
    -webkit-touch-callout: none;
    -webkit-user-select: none;
    -khtml-user-select: none;
    -moz-user-select: none;
    -ms-user-select: none;
    user-select: none;
}
```

[52] They will look for `jQuery` and `$` in the global namespace and extend them.

[53] For example, currently, the STF tests written by Cikryt depend on a browser-widget with `JQueryExtension` [Cik15, p.62].

Listing 13: Prevent default context menu on right click

```
// Globally, disable default context menu.
$$(document).on('contextmenu', function(e) {
    e.preventDefault();
});
```

Using the Application Artefact for Multiple Pages

The Saros GUI is divided into multiple pages, each of them is basically a self-containing application, completely independent from the others. Currently, there is the main page and one page for each wizard. They are independent because each is running in its own browser instance and there is no direct way to communicate between pages. But the pages share functionality and models like the contact list are not only necessary for one page. It is nice to have one build artefact, at all, instead of one per page since a lot of modules are shared anyway. When using one bundle for all the pages there must be some possibility to tell the application which page is needed, initially. This is done by setting the `page` property in the HTML file, for an example on the basis of the main page see Listing 14. Each valid value of `page` must be handled in `html/js/app.js` via referring to the corresponding page, see Listing 15.

Listing 14: Set `page` property in the head of the HTML file

```
<script>
    // Decide which page to render.
    window.app.extend({
        page: 'main-page'
    });
</script>
```

Listing 15: Show page with respect to the `page` property in `html/js/app.js`

```
switch (this.page) {
    case 'main-page':
        new MainPage({
            el: appContainer,
            model: this.state
        });
    break;
    ...
}
```

Handling of DOM-Events

There seems to be an issue with the propagation of DOM events when running the application inside the browser-widget which occurred when opening the context menu (provided by an external Bootstrap plugin, see `html/js/vendor/bootstrap-contextmenu.js`). When the application is running in a browser, stand-alone, everything works as expected. When

running it inside an IDE, the context menu is shown on the `mousedown` event, as expected, but on the following `mouseup` event it is removed again, instantly. Debugging revealed, that the property `which`[54] of the jQuery event object is not working correctly[55], which may be in relation to this issue. The problem is fixed via introducing an offset when showing the context menu such that the cursor is focusing the context menu element.

5.4 Tooling

In this section, the necessary tools for building and testing are introduced. As already mentioned, this step is important to ease the development process for new and unexperienced developers. Tools can support the developer to find errors early (precompilation of templates, see 5.4.1, unit testing, see 5.4.2, linting, see 5.4.3), ease the review process (code auto-formatting, see 5.4.4) and accept responsibility for dependency management (5.4.1). As already mentioned in Section 4.5, the environment for those tasks is NodeJS. All tasks are defined in `html/package.json` in the `scripts` object and can be run via `npm run <scriptName>`.

5.4.1 Building

The essential task for building the application is `build` which just calls two other tasks one after the other, `build:jade` and `build:js`, which are described in the following:

build:jade In this task, each JADE template is precompiled into a JavaScript function via *templatizer*[56], a NPM package which gets a directory as input argument, looks for `*.jade` files in this directory and generates a CommonJS module which provides an object with a property for each JavaScript function, named after the `*.jade` file. In the Saros GUI, this module is present in `html/js/templates.js`.

build:js In this task, the CommonJS modules are bundled into one JavaScript file via Browserify[57]. Browserify is a tool for bundling applications based on CommonJS modules. As input argument the entry file of the application is given, in the case of the Saros GUI that is `html/js/app.js`. Browserify is than following the dependency graph defined via the required modules and bundles them into one JavaScript file which then contains the whole application. Furthermore, it is able

[54] http://api.jquery.com/event.which/
[55] https://sourceforge.net/p/dpp/bugs/858/
[56] https://github.com/HenrikJoreteg/templatizer
[57] http://browserify.org/

to create a source map with which the code in the bundle can be related to the module it is coming from to ease debugging, as described in Section 5.3. By default, the content of the original module file and some meta-data like path and filename are included in the bundle which leads to an increased file size of the bundle by at least factor 2. Thus, the tool *exorcist*[58] is utilised to outsource the source map in a dedicated file to keep the bundle small.

5.4.2 Testing

For unit tests, Mocha[59] is utilised. To run the unit tests, run `npm run test`. The directory `html/test` will be searched and available unit tests will be executed. This enables the developer to separate the unit tests in multiple files, for example according to the tested module. It is possible to specify different output formats to make the unit tests processable by a continuous integration system.

5.4.3 Linting

Linting is the process of looking for the usage of suspicious constructs in code which are likely to introduce bugs [Wik]. For example, in JavaScript, the declaration of variables without the `var` keyword is possible, however, they are moved into the global variable scope which can lead to difficult to predict side-effects. A more detailed look at such suspicious language constructs specifically in JavaScript is given by Douglas Crockford in his well-known Book *JavaScript: The Good Parts* [Cro08]. For the Saros GUI JSHint[60] is utilised, which is runnable via `npm run lint`. Such a tool can be especially helpful for developers without much experience in JavaScript to point out bad practices.

5.4.4 Code Auto-Formatting

To ease the utilisation of a common code formatting style in a project, tools to auto-format code should be utilised. To meet the code rules of the general Saros project[61], JSCS[62] is utilised to be able to do auto-formatting in the JavaScript application.

5.5 Build Integration

Analogue to the integration of the `ui` module introduced by [Cik15, p.68], the `ui-frontend` OSGi module had to be integrated in the Saros build process

[58] exorcist
[59] http://mochajs.org/
[60] http://jshint.com/
[61] http://www.saros-project.org/coderules#before_committing
[62] http://jscs.info/

in three places: in both IDEs and in the Jenkins build. Furthermore, since the `ui-frontend` module contains a JavaScript application, there is one additional build process.

5.5.1 Building the OSGi Module inside IntelliJ and Eclipse

Because the JavaScript application is built stand-alone, currently, the build of the `ui-frontend` module is straight-forward. Appropriate `.project`, `build.properties` and `MANIFEST.MF` configuration files for Eclipse as well as `de.fu_berlin.inf.dpp.ui.frontend.iml` and `modules.xml` configuration files for IntelliJ are provided. The `build.properties` respectively the `de.fu_berlin.inf.dpp.ui.frontend.iml` specifies which files will be included in the binary build.

In addition, the configuration of the `ui` module had to be adapted to specify that it has access to the resources provided by `ui-frontend`. To facilitate that, `MANIFEST.MF` respectively `de.fu_berlin.inf.dpp.ui.iml` had to be configured, accordingly.

5.5.2 Configuring the Jenkins build

For the `ui-frontend` module a new Jenkins job was created for executing quality assurance tasks. The previously presented tasks for building the application and for running the unit tests are executed. However, there are two different Jenkins jobs, one is triggered by changes pushed to Gerrit[63] and one is triggered by changes on the master branch. Currently, only in the first scenario the job for quality assurance for the JavaScript application is triggered.

5.5.3 Discussion about the Integration of the JavaScript Build Process

As said, the JavaScript build process in not integrated in the general build process of Saros, yet. I initiated a discussion on the mailing list about the JavaScript build process, in general, and whether to integrate it in the Saros build process[64]. In the following I briefly summarise the arguments for and against a separation.

Pros:

1. Only developers who care about the HTML/JavaScript part (developers who actually work on that part) must provide the environment for the JavaScript build.

[63] http://saros-build.imp.fu-berlin.de/gerrit/
[64] http://article.gmane.org/gmane.comp.ide.eclipse.saros.devel/1401

2. Developers are forced to deal with the build process and related tasks like unit testing, which should be done anyway.

Cons:

1. More complicated for new developers since process has to be understood.

2. A lot of overhead when you only want to make a small change.

However, the first argument for the separation is not valid in the current scenario. I suggested to commit the build artefact of the JavaScript application, thus, developers would not have to build the JavaScript application themselves, which has several major disadvantages, for example very huge commits in terms of diffs (differing lines between commits). Interestingly enough, in the discussion nobody expressed concerns based on that factor, however, with the first commit of the build artefact the discussion started. Therefore, no build artefacts are committed and every developer which wants to see the effects of commits containing changes in the JavaScript application, must build the application. An integration of the JavaScript build process should be investigated in the future.

5.6 Accompanying Refactorings

Due to the dependency to the Java-to-JavaScript interface, in the course of this thesis I had to make adaptions in the `ui` module, the Java-part so to speak, from time to time. One example for my work on the Java-part is a refactoring regarding the handling and propagation of information from Java to the Saros GUI[65]. Before this refactoring, the interface from Java to JavaScript was an accumulation of JavaScript functions which were responsible for the propagation of very small units of information in a dedicated way. Those JavaScript functions were called from Java with respect to the current state of the application. For example, see the function responsible for updating the connection state in the Saros GUI before the refactoring in Listing 16. Furthermore, the classes responsible for the propagation of such information (`Renderer` classes, for more information see [Cik15, p.45] and [Boh15, p.35]) were not separated, cleanly. For example, the code in Listing 16 was present in the `ContactListRenderer`, one can argue whether that is reasonable, at least. Thus, I decided to narrow the interface and to encapsulate information with respect to their domain. A `StateModel` was introduced which encapsulates information about the connection state, the active account and the corresponding list of contacts in one object. The `StateRenderer` is responsible for propagating the information held by the `StateModel` to the Saros GUI, which is done by one function call. Thus,

[65] http://saros-build.imp.fu-berlin.de/gerrit/#/c/2302/

the interface is cleaner and less arbitrary, the downside is that every time the `StateModel` changes, the whole information must be propagated to the Saros GUI. However, that is not a problem because calling the interface in the Saros GUI is just a JavaScript function call instead of HTTP-requests (as in a common web application), for example.

Listing 16: Old Java-to-JavaScript interface to propagate the connection state

```
private synchronized void renderConnectionState(
    IJQueryBrowser browser) {
     switch (connectionState) {
     case CONNECTED:
          browser.run("SarosApi.trigger('setIsConnected
              ', " + true + ");");
          break;
     case NOT_CONNECTED:
          browser.run("SarosApi.trigger('setIsConnected
              ', " + false + ");");
          break;
     case CONNECTING:
          browser.run("SarosApi.trigger('setIsConnecting
              ');");
          break;
     case DISCONNECTING:
          browser.run("SarosApi.trigger('
              setIsDisconnecting');");
          break;
     default:
          break;
     }
}
```

5.7 Results

In this section an overview is given about implemented and open functionalities of the three most important and fundamental parts of the Saros GUI: the main view, the wizard for session invitations and the wizard for joining a session. These individual components and their features of the old Saros GUI and there functionalities will be presented on the basis of the Eclipse GUI analysis by Bohnstedt [Boh15, p.17]. For each component implemented functionalities and open tasks will be enumerated. Mainly, the reason to not implement a specific feature were limitations in time or a missing data interface.

5.7.1 Saros main view

Implemented

Connecting/Disconnecting Selecting an account, connecting, disconnecting works. The connection state is visualised to the user, accordingly. Functionalities are enabled/disabled according to the connection state.

Managing Contacts Adding, renaming and deleting contacts works. Available contacts are shown in a list, their online status is visualised.

Starting a session A button to start the session wizard is available.

Open tasks

Managing Accounts Creating, deleting and editing accounts is not implemented. It is currently implemented in the IDE-specific part of Saros (Saros properties in Eclipse) and it may be reasonable to keep it there since there is no need to move it to the main view.

Invocation of Saros properties window The Saros properties window is IDE-specific and there is no functionality to call it from the `ui` module, yet. In future, this feature should be implemented.

Session management Since there is currently no interface for providing data about a running session to the Saros GUI (and it is unclear how exactly such data will look like) this part is not implemented, yet. Further functionality of this part is activating/deactivating the follow mode and visualising awareness information like opened files by other contacts, etc.

Chat The chat functionality is not available from the `ui` module at all, currently. Therefore, no work was done in that direction.

Shortcut for sharing files with a specific contact Currently, there is no feature for sharing projects with a contact when right-clicking the contact. That is a reasonable feature which is implemented in the old Saros GUI.

5.7.2 Sesseion-Invitation Wizard

Implemented

Starting a session The wizard is fully functional. It is possible to select/deselect projects/files to work on and contacts to share with.

Open tasks

Validation Currently, it is possible to start a session without specifying any files, this should not be possible

Additional wizards While waiting for other contacts to join the session and to synchronise files, a wizard with a progress bar is shown in the old Saros GUI. This wizard is not implemented yet.

5.7.3 Join-Session Wizard

Implemented

UI template To be able to test the feature in the course of the usability tests I hard-coded the interface of the wizard with some basic functionality.

Open tasks

Solid implementation The wizard must be implemented from scratch, when the data interface is specified. However, parts of the template can be reused.

Additional wizards During the process of the invitation there are wizards to show the progress of the invitation process in the old Saros GUI. Those additional wizards are not implemented, yet.

The current state of the implementation is a good foundation to finish the open tasks. A lot of effort was put in the groundwork of the application to ease the further development. The biggest open tasks are the representation of a running session (and the visualisation of corresponding informations) and the wizard for joining a session.

5.7.4 Documentation

As mentioned before, decreasing the entry hurdle for future developers is an important goal. A comprehensible application structure and architecture as well as tooling support are steps in that direction. Another very important point is a good documentation of source code. Thus, I will update and extend the existing documentation for extending the HTML GUI[66]. Furthermore, I will provide a commented tour through the JavaScript code of the Saros GUI based on the idea of JTourBus[67] to be able to comprehend the processes in and between the individual components.

[66] http://www.saros-project.org/html-gui
[67] www.saros-project.org/jtourbus

6 Applying the User-Centered Design Process

Since it is sometimes difficult to find test users, especially short-term, I tried to plan the iterations early to have a solid test user base. However, estimating the effort for the implementation of the essential features is difficult. The interdependency between Bohnstedt's and my thesis made it even more insecure. It was intended to start the usability tests when the new Saros GUI is in a state where it is possible to start and join a session in a minimal setup. This would have required the corresponding wizards and the underlying interface between Java and JavaScript. However, as Bohnstedt states, the implementation of the interface was a lot more cumbersome as estimated [Boh15, p.28].

Since the defined requirements (see Section 3.4.1) for a usability test were not satisfied at the time of the first iteration, I decided to split it into two iterations. The first iteration (see Section 6.1) was done with the old Saros GUI. That gave me the possibility to do a trial run of the UCD process and, eventually, identify usability problems in the old GUI to avoid the transfer of old usability problems into the new one. Shortly after that, the initial iteration with the new Saros GUI was done. Between both iterations I was able to mock essential parts of the GUI to have a reasonable amount of features to test (see Section 6.2 for details).

The heuristic evaluation (see Section 6.3) was done delayed for the same reason as the initial iteration of the new Saros GUI. After the initial iteration and the heuristic evaluation I fixed a subset of identified usability problems before starting the final iteration (see Section 6.4). In each iteration, each test session followed the process described in 3.4.2. During the overall process 19 usability problems were identified, for 7 of them I committed fixes. The next sections explain the process of each of the mentioned steps and present the results in more detail.

6.1 Preliminary Iteration with the old Saros GUI

Originally, the initial iteration was planned with four users. Due to the reasons mentioned above, I split the iteration into two iterations with two users each.

Test Users Two users, both without experience with Saros (however, the concept and use-case of Saros was known) and with high respectively medium experience with Eclipse.

Setup I as well as the test user had a laptop with Eclipse and Saros with the old GUI installed and running. Furthermore, there was screen and audio

recording on the user's laptop. The given tasks as well as the schedule of the test session are available in Appendix A.1.

Results My expectations were low before testing the old GUI as I saw the test sessions as trial runs for the following iterations. Both test users were able to complete all tasks. The programming task (*Task 2c*, A.1) was for loosen up the atmosphere and getting to know Saros and did not contribute anything to the results.
However, in addition, some minor usability problems were identified. For example, both users had problems with finding buttons for specific functionalities like connecting or leaving a session (due to icon-only buttons). Table 5 lists all usability problems found in this iteration.

Table 5: Usability problems found in preliminary iteration with the old Saros GUI. See A.3 for details.

No	Title	Fatality
#1	Unclear state of contacts	2
#2	Functionality hidden behind icon-only buttons	2
#3	Unclear when follow mode stops	2
#4	Moving contacts in Session container is confusing	3

6.2 Initial Iteration with the new Saros GUI

Between the preliminary iteration and the initial iteration with the new Saros GUI, I mocked the wizards for starting and joining a session[68]. Therefore, in comparison to the previous iteration, I was not able to setup a real Saros session, but it was sufficient to have one laptop with a running Eclipse instance with Saros. Testing the wizards was possible, reasonably: After finishing the wizards for starting and joining a session nothing happens, which was not of importance for the test. I started the wizard for joining a session via a hidden button and explained the authentic process to the test users, afterwards the wizard could be tested, as usual.

Test Users Two users, both without experience with Saros (they heard of the software for the first time) and medium experience in Eclipse.

Setup I sat next to the test user but tried to stay in the background, however, at the given time I had to trigger the wizard to join a session, manually. Furthermore I gave some additional information to the user to explain him the scenario, in which the wizard would be triggered, practically. Upfront,

[68] Mostly, the UI was hardcoded in HTML with some basic validation checks in JavaScript.

I needed to give some more information about Saros and its concepts, for example what exactly a session is and what belongs to it, etc. The Saros GUI functionality was as described in Section 5.7. The given tasks as well as the schedule of the test session are available in Appendix A.2.

Results Despite the lack of experience with Saros and its concepts, the entrance in the application and its functionalities went well. However, in the course of this iteration, a lot of minor issues were identified since it was the first test for the new Saros GUI. Table 6 lists all usability problems found in this iteration.

Table 6: Usability problems found in initial iteration with the new Saros GUI (only previously unknown problems are listed). See A.3 for details.

No	Title	Fatality
#5	Missing default account selection	2
#6	Missing auto-connect feature when selecting an account	2
#7	Missing default value in input fields	2
#8	Contacts looking like links, action expected	2
#9	Default context menu of the browser shown on right-click	2
#10	Contact container is not collapsible	1
#11	No default actions on enter in dialogs	2
#12	No Work together on... shortcut available	2
#13	Meaning of tabs in wizard for joining a session unclear	4

6.3 Heuristic Evaluation

The heuristic evaluation was done after the first iteration with the new Saros GUI because the essential functionality was not available, before. Because of the narrow time schedule, there were no fixes to usability problems in the meantime.

Evaluator The evaluator was a researcher of the Software Engineering working group who is familiar with Saros and its concepts.

Setup I sat next to the evaluator, who was freely exploring the UI and the functionalities. In interesting situation we discussed the feature or the problem in more depth.

Results most of the identified problems from the initial iteration were confirmed, however, one new problem was found, as well. Table 7 lists all usability problems found during the heuristic evaluation.

Table 7: Usability problems found in the heuristic evaluation (only previously unknown problems are listed). See A.3 for details.

No	Title	Fatality
#14	Inconsistencies in button ordering (wizards)	1

6.4 Final Iteration

Before starting the final iteration, improvements were made in the Saros GUI according to the results of the previous iterations and the heuristic evaluation (see A.3 for details).

Test Users Four users, three of them were former or current participants of the Saros project. Therefore, they know the functionalities and the old Saros GUI, but they had no experience with the new UI. The other user participated the second time in this UCD process (he already take part in the preliminary iteration, see Section 6.1).

Setup As in the initial iteration, see 6.2.

Results This iteration was particularly interesting since all participants had experience with the old Saros GUI. For example, it was interesting to see how well the new GUI fits the conventions and workflows of the old one. Furthermore, since they were familiar with the functionalities, I had the impression that the tests were more intense. In addition, this iteration asses the fixes made until then. As expected, it turned out that some solutions did not really fix a problem, other did. Table 8 lists all usability problems found in this iteration.

6.5 Summary

The most important and beneficial methodology was testing with real users, most usability problems were found that way. The heuristic evaluation confirmed a lot of findings and even one additional problem was identified. The interviews after the test session were not so important to identify usability problems, however, in the discussion ideas for solving problems evolved. Video and audio recording were interesting for retracing particular problems and corresponding discussions. Furthermore, as the experimenter, one had not enough time to take notes covering all problems and statements of the user, thus the recordings helped to document the findings, thoroughly.

Table 8: Usability problems found in final iteration with the new Saros GUI (only previously unknown problems are listed). See A.3 for details.

No	**Title**	**Fatality**
#15	Specific input fields in dialogs should be focused, initially	1
#16	Unclear semantics of connect/disconnect	2
#17	Complicated to create an new account	2
#18	Unsorted contacts / no way to sort contacts	1
#19	CMD+A/CTRL+A not working in input fields in dialogs	1

7 Conclusion

7.1 Results

In the course of this thesis I implemented a new Saros GUI based on HTML and JavaScript utilising a UCD process to maximise the usability of the GUI. I gave an introduction to related work on Saros (Section 2) and to fundamental terms and concepts of the field of usability engineering and UCD (Section 3.1). Based on that, I elaborated an UCD process to iteratively gather feedback from real users to be able to incorporate insights about the usability of the GUI into the process of implementation (Section 3.2 to 3.5). After three iterations of usability tests (the first iteration was still done with the old Saros GUI) and an heuristic evaluation, 19 usability problems were identified and 7 of them fixed. Details about the results of the UCD process are given in Section 6.

As groundwork for the implementation I introduced essential technologies and relevant previous works on Saros (Section 4.1 to Section 4.3). I identified special requirements of the application and, based on that, evaluated possibly suitable JavaScript frameworks to build on (Section 4.4). AmpersandJS was chosen because it offers the best compromise between provided features and a low entry hurdle for new developers. Next, I presented the chosen environment for tool support for the JavaScript development process (Section 4.5).

Based on this technologies I implemented a new Saros GUI, on which I had a detailed look in Section 5. After presenting the extended module structure in Eclipse's OSGi context (Section 5.1), I introduced individual components and their interdependencies (Section 5.2), afterwards I presented special challenges (Section 5.3), details of the build process (Section 5.5) as well as further tooling (Section 5.4) and accompanying refactorings (Section 5.6). During the implementation I tried to support a convenient entrance for future developers, even without experience in web development, however, not of costs of good architecture and best practices.

In Section 5.7, I summed up the results of the implementation. In my opinion, a solid groundwork was laid for future developers to work on the Saros GUI and to implement missing features to be able to replace the old Saros GUI in the future.

7.2 Future Work

Although the foundation for a Saros GUI based on HTML and JavaScript is built with the implementation in the course of this thesis, there are a lot of open tasks to take care of before the GUI is ready to replace the old Saros

GUI. Regarding functionality, these open tasks are summed up in Section 5.7. Since the utilisation of a UCD process was quite a success, I find it reasonable to suggest that further implementation should be accompanied with a similar process, if possible, too. Furthermore, there is some work to do regarding the build of the JavaScript application. The build process should be integrated in the general Saros build process as well as in the Jenkins jobs, see Section 5.5 for more details.

As Bohnstedt [Boh15, p.62] already pointed out, there is also some work to do in the direction of test coverage, on the Java-side as well as on the JavaScript-side. In addition, the extension of the STF framework for testing the HTML GUI by Cikryt [Cik15, p.61] is neglected so far.

A Appendix

A.1 Usability Test Task Sheet 1

Saros Usability Evaluation

1 Process

- Welcome
- Setup and Introduction of the Environment
- Usability Evaluation
- Interview
- Debriefing

The entire evaluation process will take approx. 60 minutes.

2 Usability Evaluation - Tasks

1. Basics
   a) Connect with your XMPP account
   b) Add *mustermann@saros-con.imp.fu-berlin.de* to your contacts
   c) Add *bzums@saros-con.imp.fu-berlin.de* to your contacts
   d) Give the user *bzums@saros-con.imp.fu-berlin.de* a nickname
   e) Delete *mustermann@saros-con.imp.fu-berlin.de* from your contacts
2. Incoming Session Negotiation
   a) Accept the incomming session request: create a new project for *FizzBuzz*
   b) Open the file the contact *bzums@saros-con.imp.fu-berlin.de* has opened in the session
   c) Solve *FizzBuzz*
   d) Leave session
3. Outgoing Session Negotiation
   a) Partially Share the project *de.fu_berlin.inf.dpp.ui.frontend* with contact *bzums@saros-con.imp.fu-berlin.de*: share only the directory *html/js*
   b) Enter the follow mode: start following *bzums@saros-con.imp.fu-berlin.de*
4. Incoming Session Negotiation with Multiple Projects
   a) Accept incomming session request: create a new project for *de.fu_berlin.inf.dpp.ui* and use the existing one for *de.fu_berlin.inf.dpp.ui.frontend*

1

A.2 Usability Test Task Sheet 2

Saros Usability Evaluation

1 Process

- Welcome
- Setup and Introduction of the Environment
- Usability Evaluation
- Interview
- Debriefing

The entire evaluation process will take approx. 45 minutes.

2 Usability Evaluation - Tasks

1. Basics
   a) Connect with the account *test-account-upb@saros-con.imp.fu-berlin.de* (password: *test-account*)
   b) Add *mustermann@saros-con.imp.fu-berlin.de* to your contacts
   c) Add *bzums@saros-con.imp.fu-berlin.de* to your contacts
   d) Give the user *bzums@saros-con.imp.fu-berlin.de* a nickname
   e) Delete *mustermann@saros-con.imp.fu-berlin.de* from your contacts
   f) Disconnect
2. Incoming Session Negotiation
   a) Connect with the account *anotherone@saros-con.imp.fu-berlin.de* (password: *another*)
   b) Accept the incomming session request: create a new project for *de.fu_berlin.inf.dpp.saros* and use the existing one for *de.fu_berlin.inf.dpp.ui.frontend* and *de.fu_berlin.inf.dpp.ui*
   c) Disconnect
3. Outgoing Session Negotiation
   a) Connect with the account *test-account-upb@saros-con.imp.fu-berlin.de* (password: *test-account*)
   b) Start a session and partially share (share only the directory *html/js*) the project *de.fu_berlin.inf.dpp.ui.frontend* with the following contacts:
      - *bzums@saros-con.imp.fu-berlin.de*
      - *basti-test-account@saros-con.imp.fu-berlin.de*
      - *peter-post@saros-con.imp.fu-berlin.de*

A.3 Catalogue of Usability Problems

#1	**Unclear state of contacts**	**2**
Only online contacts are marked explicitly with an *online* tag. When no contact is online, there is no indication about their state.		
Source: Usability Tests		
Approaches: • Provide a tag for every possible state and show it permanently		
Discussion: This problem was found in the old Saros GUI and was not fixed in the initial design of the new Saros GUI, thoroughly. Afterwards, the problem was fixed with the following commit: http://saros-build.imp.fu-berlin.de/gerrit/#/c/2791/		

#2	**Functionality hidden behind icon-only buttons**	**2**
Users struggled to find buttons for basic functionalities like connecting, disconnecting or leaving a session.		
Source: Usability Tests		
Approaches: • Add labels to certain buttons instead of using icons, only		
Discussion: This problem was found in the old Saros GUI. The new Saros GUI was designed, accordingly.		

#3	**Unclear when follow mode stops**	**2**
Users expect to stop following another contact when start typing. That is not the case. Furthermore, users did not realise that the follow mode stopped on other occasions.		
Source: Usability Tests		
Approaches: • Give the user explicit feedback when the follow mode is stopped • Explicitly explain to the user when the follow mode stops		
Discussion: This is a problem found in the old Saros GUI and can not be fixed by UI adaptions, only.		

#4	**Moving contacts in *Session* container is confusing**	**3**
In the old Saros GUI contacts participating in a session are moved from the list of contacts to the list of contacts in a session. This is confusing, because they are still usual contacts and should be present in the list of contacts.		
Source: Usability Tests		
Approaches: • Instead of moving contacts from one list to another, copy them		
Discussion: This is a problem found in the old Saros GUI and should be fixed in the new one. However, currently the *Session* view is not implemented.		

#5	**Missing default account selection**	**2**
When starting Saros, there is no default account selected, instead, the message *No account selected* is shown to the user. Furthermore, the current solution is inconsistent with the old GUI, where an default account is selected.		
Source: Usability Tests		
Approaches: • Select a default account, for example the last recently used		
Discussion: Since this fix would also include an extension of the data interface (the last recently used account must be provided, for example), I did not approach this problem in the course of this thesis.		

#6	**Missing auto-connect feature when selecting an account**	**2**
When selecting an account, users expect to connect, instantly. Furthermore, that would be consistent with the old Saros GUI.		
Source: Usability Tests, Heuristic Evaluation		
Approaches: • When selecting an account, auto-connect with that account		
Discussion: The problem was fixed with the following commit: http://saros-build.imp.fu-berlin.de/gerrit/#/c/2818/		

#7	**Missing default value in input fields**	**2**
Currently, the user has to type manually, for example when using an existing project in the wizard for joining a session. However, with high probability, the name is the same as the shared project and could therefore be provided as default value.		
Source: Usability Tests, Heuristic Evaluation		
Approaches: • Provide default values in input fields wherever possible		
Discussion: The problem is easy to fix when the wizard for joining a session is implemented in a stable way (currently it is just mocked).		

#8	**Contacts looking like links, action expected**	**2**
In the Saros main view, when hovering over contacts, they are highlighted and the cursor changes to *pointer*. Therefore users expect an action on left-click.		
Source: Usability Tests, Heuristic Evaluation		
Approaches: • Avoid *pointer* cursor • Add action on left-lick, for example inline editing of the nickname		
Discussion: The idea of explicitly hovering a contact was to make clear that there is some action available, in this case on right-click. Thus, I would not disable the hovering effect, but maybe keep the usual *arrow* cursor. Providing an action of left-click would be another possibility. However, it is unclear what the user expects and it may lead to significant development effort depending on the action.		

#9	**Default context menu of the browser shown on right-click**	**2**
On right-click, the default browser context-menu is shown with options like *reload*, which makes no sense and confuses the users.		
Source: Usability Tests, Heuristic Evaluation		
Approaches: • Prevent showing the default context menu		
Discussion: The problem was fixed with the following commit: http://saros-build.imp.fu-berlin.de/gerrit/#/c/2857/		

#10	***Contact* container is not collapsible**	**1**
The list of contacts leads to vertical scrollbars, quickly. Furthermore, users expected to be able to collapse the container.		
Source: Usability Tests, Heuristic Evaluation		
Approaches: • Trigger collapse of *Contacts* container on left-click		
Discussion: The problem was fixed with the following commit: http://saros-build.imp.fu-berlin.de/gerrit/#/c/2819/		

#11	**No default actions on enter in dialogs**	**2**
Clicking enter in dialogs triggers closing the dialog, however, without actually performing an action. That is especially confusing because it is not clear to the user whether or not the action was performed. Users expect that the corresponding action is triggered, for example adding/renaming a contact.		
Source: Usability Tests, Heuristic Evaluation		
Approaches: • Trigger default action on enter		
Discussion: The problem was fixed with the following commit: http://saros-build.imp.fu-berlin.de/gerrit/#/c/2820/ However, in dialogs for confirming deletion, currently nothing happens on enter to avoid accidental deletions.		

#12	**No *Work together on...* shortcut available**	**2**
Like in the old Saros GUI, users expect to be able to share projects/files with a specific contact via right-clicking on the contact and choosing *Work together on...*.		
Source: Usability Tests, Heuristic evaluation		
Approaches: • Provide *Work together on...* in the context menu on right-click on the contact		
Discussion: Clicking on *Work together on...* should open the wizard for starting a session with the specific contact selected, by default. This feature should be implemented, especially because expert users expect it.		

#13	**Meaning of tabs in wizard for joining a session unclear**	**4**
Currently, in the join session wizard, multiple incoming projects are visualised in tabs. First, some users had problems with identifying the tabs, at all. Second, it was not clear that each tab is representing one project. Since the current default selection input elements is valid, the user is not encouraged to navigate to each tabs and explore them but tends to just click *Finish*.		
Source: Usability Tests		
Approaches: • Visualise the tabs as tabs, mor explicitly • Provide additional information about the semantic of the tabs to the user (info text, tooltips)		
Discussion: Maybe it helps the tabs vertically to mark them as tabs, more explicitly. When the wizard is implemented fully, this issue should be investigated further, best with utilising further usability tests.		

#14	**Inconsistencies in button ordering (wizards)**	**1**
Buttons are ordered such that the confirming button is not the right most. However, in in the old GUI and in Eclipse, that is a convention.		
Source: Usability Tests, Heuristic Evaluation		
Approaches: • Rearange the buttons		
Discussion: The problem was fixed with the following commit: http://saros-build.imp.fu-berlin.de/gerrit/#/c/2858		

#15	**Specific input fields in dialogs should be focused, initially**	**1**
When a user opens the dialog for renaming a contact, for example, it would be more convenient to auto-focus the corresponding input field since the user opens that specific wizard with the intention to edit the nickname of a user. Now, the user has to click on the input field, first. In addition, maybe it is reasonable to preselect the value in the input field.		
Source: Usability Tests		
Approaches: • Auto-focus input field in certain dialogs • Preselect value of certain input fields in dialogs		
Discussion: I think this is an reasonable enhancement. There is no fix, yet, due to time issues.		

#16	**Unclear semantics of connect/disconnect**	**2**
The labels *Connecting* and *Disconnecting* are confusing to some users. Maybe *Login* and *Logout* are more clear since they are known from many online platforms.		
Source: Usability Tests		
Approaches: • Rename labels		
Discussion: This issue should be investigated further.		

#17	**Complicated to create an new account**	**2**
Independent of the new Saros GUI: it is currently not very clear how to create an new account because it is not possible to do so in the IDE-specific Saros properties menu.		
Source: Usability Tests		
Approaches: • It should be possible to create an account in the IDE-specific Saros properties menu		
Discussion: This is a general and GUI-unspecific problem.		

#18	**Unsorted contacts / no way to sort contacts**	**1**
Currently, the order of contacts is not guaranteed. Users want to be able to sort contacts or expect them to be sorted according to a least-recently-used strategy.		
Source: Usability Tests		
Approaches: • Sort users by default based on least-recently-used • Enable sorting based on different properties (nickname, least-recently-used)		
Discussion: The interface must be extended for this feature. I think it would be reasonable to implement a default sorting according to least-recently-used.		

#19	**CMD+A/CTRL+A not working in input fields in dialogs**	**1**
Users want to select the whole value of an input field via the named shortcuts which is currently not working.		
Source: Usability Tests		
Approaches: • Implement the described behaviour		
Discussion: Not implemented, yet, due to time issues.		

References

[Ang] AngularJS. AngularJS Developer Guide. https://docs.angularjs.org/guide/introduction#! Retrieved August 27th, 2015.

[Boh15] Matthias Bohnstedt. Entwicklung einer IDE-unabhängigen Benutzeroberfläche für Saros. Master's thesis, Freie Universität Berlin, 2015.

[Cik15] Christian Cikryt. Evaluating the Use of a Web Browser to Unify GUI Development for IDE Plug-ins. Master's thesis, Freie Universität Berlin, 2015.

[Cro08] Douglas Crockford. *JavaScript: The Good Parts*. O'Reilly Media, Inc., 2008.

[Dur14] Damla Durmaz. Verbesserung der Action Awareness im Open Source Plug-in Saros. Master's thesis, Freie Universität Berlin, 2014.

[Int98] International Organization for Standardization. ISO 9241-11: Ergonomic requirements for office work with visual display terminals (VDTs) - Part 11 : Guidance on usability. Technical report, International Organization for Standardization, Geneva, Swiss, 1998.

[Int99] International Organization for Standardization. *ISO 13407. Human Centred Design Process for Interactive Systems*. Geneva, Swiss, 1999.

[Kah11] Björn Kahlert. Verbesserung der Out-Of-Box-Experience in Saros mittels Heuristischer Evaluation und Usability-Tests. Master's thesis, Freie Universität Berlin, 2011.

[KCF92] Clare-Marie Karat, Robert Campbell, and Tarra Fiegel. Comparison of empirical testing and walkthrough methods in user interface evaluation. In *Conference on Human Factors in Computing Systems '92*, pages 397–404, 1992.

[MVSC05] Ji-Ye Mao, Karel Vredenburg, Paul W. Smith, and Tom Carey. The state of user-centered design practice. *Commun. ACM*, 48(3):105–109, March 2005.

[Nie93] Jakob Nielsen. *Usability Engineering*. Academic Press, London, 1993.

[Nie00] Jakob Nielsen. Why You Only Need to Test with 5 Users. http://www.nngroup.com/articles/why-you-only-need-to-test-with-5-users/, March 2000. Retrieved August 25th, 2015.

[NLM94] Jakob Nielsen and Robert L. Mack. *Usability Inspection Methods*. John Wiley & Sons, 1994.

[Sol11] Arsenij Solovjev. Evaluation der Mechanismen zum Darstellen der Workspace Awareness in Saros. Bachelor's Thesis, Freie Universität Berlin, 2011.

[Spi12] Maria Spiering. Verbesserung der Usability von Saros unter Verwendung eines User-Centered Design Ansatzes. Master's thesis, Freie Universität Berlin, 2012.

[W3C] W3C. W3C - Conformance: requirements and recommendations. http://www.w3.org/TR/REC-html40-971218/conform.html#deprecated. Retrieved August 25th, 2015.

[Wal12] Alexander Waldmann. Prüfung und Verbesserung der Usability von Saros im produktiven Einsatz. Diploma thesis, Freie Universität Berlin, 2012.

[Wik] Wikipedia. Lint (software). https://en.wikipedia.org/wiki/Lint_%28software%29. Retrieved August 29th, 2015.