Master's thesis at the Institute for Computer Science,

Software Engineering Working Group

Evaluating the Use of a Web Browser to Unify GUI Development for IDE Plug-ins

Christian Cikryt
Student ID: 4285814
c.cikryt@fu-berlin.de

Berlin, April 15, 2015

Submitted to: Prof. Dr. Lutz Prechelt
and Prof. Dr. Claudia Müller-Birn
Supervisor : Franz Zieris, M.Sc.

Abstract

Saros, an Eclipse plug-in for distributed collaborative programming, is currently being ported to the IntelliJ IDEA platform. As both IDEs use different graphical widget toolkits, SWT and Swing respectively, this results in a duplication of the GUI code. This thesis investigates whether the integration of a web browser into both plug-ins and the display of Saros in HTML-based technologies is a viable alternative. It avoids the redundant implementation of the user interface and maximises the common code base. After a thorough initial evaluation, a prototype is developed that encloses all fundamental mechanisms including GUI tests in order to qualitatively assess this approach. The prototype has been closely integrated into Saros and provides a basis for further development and evaluation. The choice and integration of the web browser constitutes a large part of this work. In the process an existing wrapper for the SWT browser has been further developed into an external library.

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 All parts taken from published and unpublished scripts are indicated as such.

Berlin, April 15, 2015

(Christian Cikryt)

Contents

1 Introduction

In every ongoing software project with a sizeable code base, decreasing future development costs will become a central point. One way to achieve this is the reduction of redundancy in the code with the goal to apply the same change in only one place. However, let us consider for a moment as an example programs that should run on multiple platforms using different graphical toolkits. Obviously, this implies duplication, not necessarily on the code level as they are after all using objects from different libraries, but the underlying logic remains the same. The standard approach to maximising the common code in this scenario is to introduce abstractions and hide all the differences behind them. However, one will still end up with different implementations for the same behaviour[1].
In this thesis, I evaluate the idea of using a browser and web technologies instead of different native toolkits for the user interface in order to unify the GUI code. I examine the concrete example of the Integrated Development Environment (IDE) plug-in Saros[2] that was originally developed for the Eclipse platform and is now ported to the IntelliJ IDEA environment[3]. Besides the support for two platforms which use different graphical toolkits, the plug-in shall be running on Windows, Mac OS, and GNU/Linux. Even though the plug-in is written in Java, the latter becomes an issue because the GUI toolkit and the browser may depend strongly on the operating system.
As we will learn during the evaluation phase, there are only a handful of viable Java web browsers and each has its limitations for our scenario. Therefore this work provides, as a byproduct, interesting knowledge about embedding a Java browser into an IDE and about the choice of Java web browsers in general.
After a thorough evaluation, an HTML-based prototype is created and used to take a close look at the resulting code structures, especially the communication between Java and Javascript. This prototype aims to implement all required mechanisms including GUI tests to detect possible problems with the browser approach and assess compatibility across operating systems.

1.1 The plug-in of interest: Saros

Saros is an Eclipse plug-in for distributed collaborative programming. It is free software and licensed under the GPL 2.0[4]. Distributed means that collaborators may

[1] Of course, parts of the GUI and the behaviour differ on each platform by choice.
[2] `http://www.saros-project.org/` (retrieved 20 March 2015)
[3] See subsection 1.5 for the introduction of both IDEs.
[4] `http://www.gnu.org/licenses/gpl-2.0.html` (retrieved 20 March 2015)

be at different physical locations while working on the same software project. Saros aims to remove resulting barriers by providing a collaborative real-time editor that highlights others' contributions and synchronising shared project files, for example. It started in 2006 as part of Riad Djemili's Diploma thesis [Dje06] and is continuously developed by members, mostly students, of the Software Engineering working group at Freie Universität Berlin.

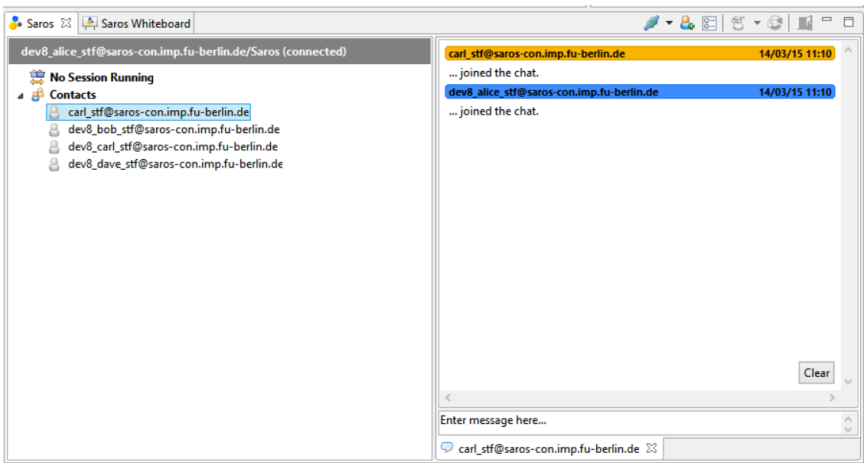

Figure 1.1: The main view of Saros in Eclipse

The Saros IntelliJ IDEA plug-in has not been released yet, but can be tested with a limited set of features consisting of the management of accounts, contacts, and the sharing of projects.

1.2 Motivation

Since the aforementioned porting to the IntelliJ IDEA platform requires the GUI to be replicated because Eclipse and IntelliJ use different graphical toolkits, the wish for a less redundant solution arose. The obvious advantage would be maintainability as future development could be happening on the same code base and changes would not have to be ported to another implementation. Furthermore, this could pave the way for a port to the Netbeans[5] platform, which is currently being evaluated with a prototype by Sabine Bender in her Bachelor's thesis [Ben15].
Initiated by browser-based widgets which Björn Kahlert, a researcher in the Software Engineering working group, wrote to display the results of his work inside Eclipse, the idea arose to use web technologies for the user interface of Saros as well. The plan is to embed a browser in the Saros plug-in on each platform and write the GUI in HTML and Javascript. Ideally, only the embedding code will differ for each IDE, while the HTML part can be re-used. Note that the Eclipse plug-in offers additional

[5] https://netbeans.org/

functionality outside the Saros view like highlighting in the editor view and extending Eclipse's configuration menu. The implementation of those features will remain IDE-specific and is outside the scope of this thesis. Figure 1.1 shows the Saros view that should be replaced by a website; this is the part of the GUI I will concentrate on. In addition, I will consider wizards, which are used for adding accounts or sharing projects for example. They are invoked when the corresponding buttons are clicked in the Saros view and currently implemented as dialogue windows.

1.3 Goals

The primary goal is to decide whether the HTML-based approach is viable and should be pursued in the future. Therefore, possible browser candidates are to be identified in a thorough evaluation. One special challenge here is to find a suitable browser for IntelliJ IDEA as the list of candidates is short. Prior to the actual implementation of a prototype it remains to be seen if this approach is indeed promising compared to the traditional one.
A prototype should be implemented to see whether all current elements of the Saros view can be displayed in the browser. Later on, the prototype is extended to take a close look at the interface between Java and HTML in order to identify possible implementation problems. The HTML prototype should also incorporate the elementary structure for GUI tests.

1.4 Terminology

Throughout this work I will use the following terms. I will use IntelliJ as a synonym for IntelliJ IDEA, since the other products and variants of the IntelliJ platform do not need to be distinguished for this thesis.
When I write "free software" I refer to the definition by the Free Software Foundation[6] and make no statement about the price. Open source software might be a more common term for this kind of software, but it has a slightly different focus.
Saros/E denotes the plain old Saros Eclipse plug-in (without HTML), while Saros/I is its IntelliJ counterpart (again without HTML).
I abbreviate user interface (UI), graphical user interface (GUI), and integrated development environment (IDE).

1.5 Introducing GUI toolkits and IDEs

1.5.1 Abstract Window Toolkit (AWT)

AWT is Java's original GUI toolkit. It provides only a thin abstraction layer around the native widgets provided by the operating system. Although AWT has been su-

[6] See `https://www.gnu.org/philosophy/free-sw.en.html` (retrieved 20 March 2015).

perseded by Swing, Swing uses AWT's interface for the communication with the operating system regarding the management of windows and events.

1.5.2 Swing

Swing was developed in order to provide a richer set of GUI components than AWT. From Java 6 Update 12 onward, Swing and AWT widgets can be mixed without AWT widgets falsely appearing over Swing widgets[7].
Swing GUI elements are pure Java with no native code. Instead of wrapping native GUI components, Swing draws its own components by using the Java 2D API to call low level operating system drawing routines. "Swing is currently in the process of being replaced by JavaFX"[8].

1.5.3 SWT

The Standard Widget Toolkit (SWT)[9] is an alternative to the Java GUI toolkits provided with Java itself, namely Abstract Window Toolkit (AWT) and its successor Swing. It is free software licensed under the Eclipse Public License (EPL) and is developed by the Eclipse Foundation closely linked to the Eclipse IDE. For the display of GUI elements, SWT calls native GUI libraries of the operating system and, consequently, the SWT implementations differ for each operating system. However, code using SWT does not have to be ported to different platforms. This is relevant for this thesis since Saros will have to deliver the SWT binary with the plug-in (see subsection 3.1.2).
According to Eclipse's FAQ[10] the motivation for the creation of SWT was to have the "native look and feel" and the "native performance" for widgets in contrast to Swing.[11]

1.5.4 JavaFX

"JavaFX is a set of graphics and media packages"[12] for the cross-platform development of rich client applications. Whenever I am talking about JavaFX in this thesis I am referring to JavaFX 2.2 and above. As of Java 7 Update 6 it is part of the Java runtime environment. It features declarative GUI development using XML and CSS. It is intended to replace Swing in the future, but for now both technologies

[7] http://bugs.java.com/bugdatabase/view_bug.do?bug_id=2169701 (retrieved 20 March 2015)
[8] Taken from http://en.wikipedia.org/w/index.php?title=Swing_(Java)&oldid=653577049 (retrieved 10 April 2015)
[9] https://www.eclipse.org/swt/ (retrieved 20 March 2015)
[10] http://wiki.eclipse.org/FAQ_Why_does_Eclipse_use_SWT%3F (retrieved 11 April 2015)
[11] According to Klemen Žagar (http://public.cosylab.com/CSS/DOC-SWT_Vs._Swing_Performance_Comparison.pdf, retrieved 11 April 2015) the actual performance of SWT is not necessarily better than the one of Swing depending on the environment.
[12] http://docs.oracle.com/javase/8/javafx/get-started-tutorial/jfx-overview.htm#BABEDDGH (retrieved 20 March 2015)

coexist and JavaFX applications can be integrated smoothly into existing Swing and SWT applications. This and the fact that it features a native, full-featured Java web browser make JavaFX relevant for Saros.

1.5.5 Event-dispatch thread (EDT)

All previous graphical toolkits are single-threaded, i. e. all manipulations of GUI objects have to take place on the so called event dispatch thread (EDT)[13]. Each toolkit has its own mechanisms for managing the EDT and letting clients execute methods on the EDT. For the rest of this work I will use EDT and UI thread (or SWT thread in SWT context) as synonyms.

1.5.6 Mixing of different toolkits

For the course of this work it is important to know that there are different Java GUI toolkits that can be embedded into one another. Figure 1.2 visualises the usage of SWT in Swing applications, whereas Figure 1.3 shows the usage JavaFX. The embedding of SWT in Swing (and vice versa) requires the so-called SWT-AWT bridge, which is included in SWT and not part of Java itself. For the technical details of the embedding, see section 3.1. In contrast, the embedding of JavaFX in SWT or Swing is an integral part of Java.

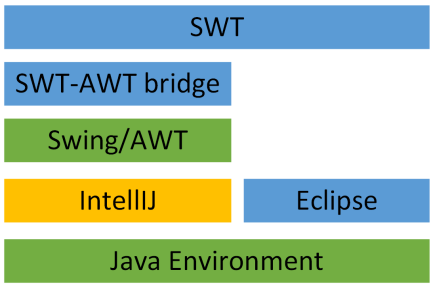

Figure 1.2: Embedding SWT in Swing and using it for both IntelliJ and Eclipse

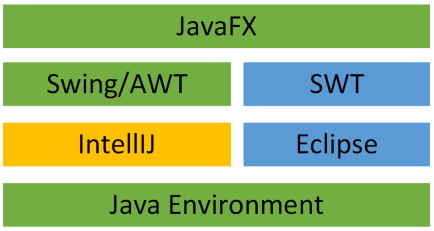

Figure 1.3: Using JavaFX embedded into Swing and SWT in IntelliJ and Eclipse

[13] The reason for this is that multithreaded GUI frameworks are extremely difficult to implement because of inherent problems with race conditions and deadlocks ([GBB+06] chapter 9.1).

1.5.7 Eclipse IDE

Eclipse[14] is an extensible IDE that can be used for software development in a large variety of programming languages. It is mainly written in Java and uses SWT. It features a workbench and a plug-in infrastructure as well as an OSGI[15] runtime environment. This is relevant as I developed a new OSGI bundle in my thesis. It is free software released and maintained by the Eclipse Foundation under the Eclipse Public License.

1.5.8 IntelliJ IDEA

IntelliJ IDEA is a cross-platform Java IDE developed by JetBrains[16] and available in a free and a commercial variant. The free Community Edition is licensed under the Apache 2.0 License[17], whereas the Ultimate Edition uses the same code base and, among others, adds support for Java EE or the Spring framework. IntelliJ IDEA is written in Java and uses Swing as its GUI toolkit.
The IntelliJ platform constitutes a generic foundation for IDEs targeting different programming languages. For this thesis, only IntelliJ IDEA is relevant.

1.6 Related work

Regarding the browser

The use of browser technologies for traditional desktop applications is not new. Google Docs and Firefox OS are just two prominent examples. With the rising popularity of smartphones and their mainly web-based services the idea to use websites instead of different native implementations for each platform became more and more relevant (even if it is mostly more of an addition than a replacement). The main difference compared to my work is that most of these web applications run headlessly and can be accessed with any web browser as opposed to a browser embedded into an application. As of now, Saros only runs embedded in the IDE so that it has to make calls into the enclosed browser to display content. In addition, the embedding scenario also requires calls from Javascript into Java to execute business methods and thus results in a bidirectional communication between Java and Javascript.
As already mentioned in section 1.2, Björn Kahlert's browser widgets[18] for Eclipse originally triggered the idea for the Saros HTML prototype. He developed those widgets to display the results of his API usability analyser[19] which is part of his PhD thesis [Kah15] in the field of qualitative data analysis. His motivation was the need for individual widgets that are easier to develop in the web context than in Java and

[14] `https://eclipse.org/` (retrieved 20 March 2015)
[15] A modular service platform for Java, see `http://www.osgi.org/Specifications/HomePage` (retrieved 20 March 2015) for more information.
[16] `https://www.jetbrains.com/` (retrieved 20 March 2015)
[17] Its source code can be found here: `https://github.com/JetBrains/intellij-community` (retrieved 20 March 2015).
[18] The browser's source code has been published alongside many other widgets on `https://github.com/bkahlert/com.bkahlert.nebula` (retrieved 20 March 2015).
[19] `https://github.com/bkahlert/api-usability-analyzer` (retrieved 20 March 2015)

exceed the performance of SWT. Section 2.2.3 describes the improvements compared to the SWT browser and explains the considerations as to how it should be used for Saros, while section 4.2 presents my modifications to it.
Damla Durmaz developed an activity log for Saros and implemented it twice, once in SWT and once in HTML, using Björn Kahlert's browser widget. Her work was a first test for the use of HTML for Saros. However, it did not include any analysis of the existing Saros GUI for an HTML re-write and did not address other IDEs or operating systems. This is where my thesis continues her work. Her experience with the HTML implementation was positive: most prominent the ease of changes to the appearance. Further details can be read in [Dur14].
The Vaadin[20] Eclipse plug-in features a UI Designer that is displayed inside a browser frame. In contrast to our use-case the Designer web application runs standalone on a remote server and there is no communication between Java and Javascript apart from initially setting the URL. The Vaadin plug-in uses the SWT browser.
There is an unofficial, outdated IntelliJ IDEA plug-in[21] of this UI Designer. This plug-in wraps the SWT Browser using Native Swing (see subsection 2.2.6), but as in the Eclipse equivalent there are no additional calls between Java and the browser, which are required for Saros if embedded into a browser. Since I was unable to find the source code of the IntelliJ plug-in, I had to take a look at the plug-in's decompiled classes for this information[22]. It is interesting to see that the Vaadin IntelliJ plug-in uses the same approach (see section 3.1): embedding the SWT browser in Swing and bundling the SWT library files in the plug-in. However, it has the same compatibility issues, e. g. with Mac OS (see section 2.2.2). Since it does not need to make calls into the browser, it does not have the need for a browser component like the one described in section 4.2. For Saros I will embed the browser directly in the IDE and make rendering calls to the browser.

Regarding the test framework

Saros Eclipse features a GUI test framework, called STF, that emulates user input via the Swing automation library SWTBot[23]. The calls to SWTBot are transmitted through Java's Remote Method Invocation (RMI) interface. STF was originally developed by Sandor Szücs in his Diploma thesis [Szü10] and enhanced by Lin Chen [Che11] and Stefan Rossbach [Ros11]. Section 4.3 describes its current structure and the adaptation for HTML GUI tests. The aim for the new HTML GUI is to make equivalent tests possible for both IDEs. Saros/I has no GUI test framework.

[20] `https://vaadin.com/home` (retrieved 20 March 2015)
[21] `https://plugins.jetbrains.com/plugin/6727?pr` (retrieved 20 March 2015)
[22] I only found out about Vaadin using the SWT browser because I explicitly (following a hint from Björn Kahlert) and repeatedly searched for it. There is no article or documentation about the embedding of a browser, I also had to explicitly look for the source code of the UI Designer which is separately developed from the rest of the Eclipse plug-in and not included in the official repositories.
[23] `http://eclipse.org/swtbot/` (retrieved 20 March 2015)

Parallel theses

Currently, there are three parallel theses in the Saros project that are of relevance to mine. Matthias Bohnstedt's Master's thesis [Boh15] evaluates the use of the HTML GUI further. He was also involved in some parts of Saros' current HTML implementation. Arndt Lasarzik is moving identified duplicated IntelliJ code into the common `core` module as part of his Bachelor's thesis [Las15]. The actual HTML implementation will be done by Bastian Sieker in his Master's thesis [Sie15]. Figure 1.4 illustrates how these areas depend on each other and relate to my work, which focuses on the UI module and the embedding part.

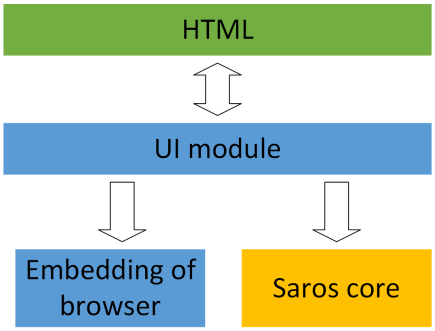

Figure 1.4: Connections between parallel theses

1.7 Outline of this thesis

Chapter 2 presents the evaluation process and results, while chapter 3 describes in detail the challenges of embedding a browser. In chapter 4, the implementation of the new HTML UI is illuminated including the development of the browser component and a draft of a graphical test framework. Before chapter 6 sums up the current state chapter 5 describes additional work I did and shares my constructive thoughts about Saros' processes.

2 Evaluation of the GUI technology

As my thesis may be seen as one part of the entire GUI evaluation process for Saros, allow me to clarify that this chapter describes the decision for the technology to concentrate on for the prototype implementation. In particular, I will explain the reasons for favouring an HTML-based solution in place of Java GUI toolkits. The working prototype serves for a more detailed examination and lays the foundation for future work.

2.1 Goals

The goals for Saros' future GUI are listed here in order to have a reference for the focus of this evaluation. They are derived from the expectations of the Saros team.

- Reduction of duplicate code / Maximisation of the common code base of both IDEs.
- Better maintainability and easier ongoing development. This is partially a result of the elimination of duplicate code, but factors like complexity and readability of the code are also playing a role here.
- Platform-independent GUI tests: the maximisation of common code should include the tests as well.

The next two items are optional goals, but should be considered in the course of this evaluation.

- Porting to further IDEs, namely Netbeans.
- Better performance than SWT or Swing: For example Damla Durmaz [Dur14] had performance problems rendering a huge number of lines in Saros' activity log using SWT[1].

2.2 Browser-based solution

As established in section 1.2, the focus of this evaluation lies on browser-based solutions since those promise less redundant code by enabling the use of the same GUI in all environments. In this section viable browser components are identified and in section 2.4 those concrete implementations are compared to native, non-browser alternatives to decide whether they are indeed favourable.
The main challenge here is to find a suitable browser for both Eclipse and IntelliJ, since each IDE uses a different graphical widget toolkit: SWT and Swing respectively

[1] This does not mean that it is impossible to render those data with acceptable response times in SWT, but it requires at least some special thoughts.

(see section 1.5). Only SWT features a browser in the toolkit itself, whereas Swing lacks an equivalent component.

2.2.1 Browsers for Swing

Before I began working on my thesis, the following four possible candidates for a Swing web browser have been listed on the Saros mailing list[2]:

1. The SWT browser embedded into Swing.
2. The commercial JxBrowser[3] written in Swing and using Chromium internally.
3. The JavaFX browser.
4. Native Swing[4].

The JavaFX browser was instantly discarded ("but it seems JavaFX is not fully supported in IDEA"[5]) with a reference to the IntelliJ plug-in developer forum. Native Swing was discarded as well because it was thought to be unmaintained ("but it is not developed since 2011 and probably lacks functionality"[5]).

I conducted additional research to make sure we did not miss any viable browser implementation. I used the two questions on the Stack Overflow website "Embed browser in Java based desktop application"[6] and "Is there a way to embed a browser in Java?"[7] mentioned on the Saros mailing list as a basis to browse related discussions on this platform. Furthermore, I asked Google for "swing browser", "embed SWT browser in swing", "java web browser" and similar queries. The results made me re-evaluate JavaFX as the quote on the mailing list did not capture the essence of the forum post. IntelliJ IDEA supports JavaFX "but the current version of IntelliJ supports running on JDK versions that don't support JavaFX, so this won't work for all users"[8]. A more detailed description of the JavaFX evaluation and its advantages can be found in subsection 2.2.5.

I also had a closer look at Native Swing aka. the DJ Project as it is in fact not unmaintained and may prove to be a helpful addition to the SWT browser. A release has not been provided since 2011 but the sole main developer is still active in discussions in the bug tracker and forum.

All other browser components I found could indeed be discarded and are listed below for reference.

---

[2] `https://lists.sourceforge.net/lists/listinfo/dpp-devel` (retrieved 20 March 2015)

[3] `http://www.teamdev.com/jxbrowser` (retrieved 20 March 2015)

[4] Project homepage: `http://djproject.sourceforge.net/ns/index.html` (retrieved 20 March 2015) and on GitHub: `https://github.com/Chrriis/DJ-Native-Swing` (retrieved 20 March 2015)

[5] `http://sourceforge.net/p/dpp/mailman/message/32789958/` (retrieved 20 March 2015)

[6] `http://stackoverflow.com/q/1454652` (retrieved 20 March 2015)

[7] `http://stackoverflow.com/q/48249` (retrieved 20 March 2015)

[8] `https://devnet.jetbrains.com/message/5507484#5507484` (retrieved 20 March 2015)

Discarded browsers

The Lobo-Project is currently not being developed and the project's homepage has been removed[9]. In addition, its supported standards are outdated (HTML 4, CSS 2). It is available on SourceForge[10].

Lobo Evolution Lobo Evolution[11] is a fork of the inactive Lobo-Project. The initial commit had just been made when I was conducting this research (18 October, 2014). As it has just one developer I do not expect a reliable implementation any time soon. At the end of my thesis (20 March 2014) I looked at the repository and homepage again. It had its first release (0.98.6) in the meantime, but still lacks any documentation and signs of users (no bug entries or discussions).

Java Chromium Embedded makes Chromium available in Java applications[12]. In contrast to the JxBrowser it is free software, but it faces the same problem of having to make sure that the Chromium binaries are present on the target system (see subsection 2.2.4). It exists since 2008 and has an active community, but it states on its homepage that it "is still very much a work in progress". Its not fully developed state was the main reason for discarding it. Furthermore, it requires Java 7 and has no 32 bit builds, whereas Saros supports Java 6 and 64 bit environments.

Webrenderer "is a pure Swing embedded Java browser component built upon Mozilla technology" (taken from its homepage)[13]. However, it is not free of charge.

JBrowser clearly states on its homepage that "JBrowser is an outdated project, I recommends to use SWT browser if possible"[14].

Summary: Possible browser components

In the next section these four remaining candidates are investigated in detail:

- SWT browser.
- JxBrowser.
- JavaFX browser.
- Native Swing.

2.2.2 The SWT browser component

The SWT browser is part of the SWT GUI toolkit, which is used by the Eclipse IDE. It is free software and exists since 2004. It is actively maintained and is, for

[9] `http://lobobrowser.org/index.jsp` (retrieved 20 March 2015)
[10] `http://sourceforge.net/projects/xamj/` (retrieved 20 March 2015)
[11] `http://sourceforge.net/projects/loboevolution/` (retrieved 20 March 2015)
[12] `https://bitbucket.org/chromiumembedded/java-cef` (retrieved 20 March 2015)
[13] `http://www.webrenderer.com/products/swing/product/` (retrieved 20 March 2015)
[14] `https://code.google.com/p/jbrowser/` (retrieved 19 February 2015)

example, wrapped in Björn Kahlert's browser component (see the end of this section on page 15). It is platform-independent but may use a different browser on each operating system because it does not ship its own rendering engine, instead accessing an installed one.
The SWT browser will run out of the box inside Eclipse, provided it can access the installed browser[15]. However, it is challenging to get it running inside IntelliJ IDEA. These challenges are described in section 3.1. Tests of the SWT browser embedded in IntelliJ on Mac OS X and GNU/Linux revealed further limitations which are presented in the next paragraphs.

Mac OS compatibility Mac OS X's graphical framework Cocoa requires the SWT event-dispatch thread to be on the application's main thread[16], otherwise leading to the immediate termination of the embedded browser. More detailed information and my solution can be found in subsection 3.1.3.
I confirmed that the embedding of SWT in Swing is broken for Java 7 and 8 on Mac OS, even when not embedded in an IntelliJ plug-in, and that Native Swing has no solution for this. The reason for the incompatibility are changes introduced with Java 7 on Mac OS[17] which affect the SWT-AWT bridge. It is undecided whether this bug should be fixed in SWT or in the Java implementation. At least for Java 7 it will not be fixed[18]. Even half a year later I could find no concrete hint that it will be fixed for Java 8.

Java version	IntelliJ	SWT
Java 6	supported	works
Java 7	unsupported	broken (will not be fixed)
Java 8	upcoming support	broken (might be fixed)

Figure 2.1: Compatibility of Java versions with IntelliJ and SWT on Mac OS

However, all of this is only relevant for the future. Currently IntelliJ IDEA requires Java 6 as runtime environment on Mac OS as "JDK 1.7 from Oracle is not officially supported yet and has known problems"[19]. Figure 2.1 summarises the compatibility of Java version with IntelliJ and SWT on Mac OS.
The first evaluation was done in October 2014, when IntelliJ 14 had just been released.

[15] I only experienced problems of this kind on GNU/Linux, see section "GNU/Linux compatibility" for more information.
[16] See `https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/Multithreading/ThreadSafetySummary/ThreadSafetySummary.html` (retrieved 12 April 2015) for more information about multithreaded programming with Cocoa.
[17] Java 6 is the last Java release maintained by Apple, all subsequent versions are provided by Oracle.
[18] `https://bugs.eclipse.org/bugs/show_bug.cgi?id=418245#c8` (retrieved 20 March 2015)
[19] `https://intellij-support.jetbrains.com/entries/23455956-Selecting-the-JDK-version-the-IDE-will-run-under` (retrieved 20 March 2015)

I re-evaluated the current state at the beginning of March 2015. Java 6 is still the only officially supported JRE version, but in the meantime Java 7 Update 40 has been released, which fixes some of the bugs. According to the bug tracker, JetBrains developers are testing support for a custom-build Java 1.8 in the minor Version 14.1[20] released on 24 March 2015, but I could find no official roadmap explaining further plans or when they plan to support Oracle's JRE version.

GNU/Linux compatibility On a freshly installed Ubuntu 14.10 the SWT browser will not work out of the box, not even in Eclipse. The reason for this is that the SWT browser currently does not work with the GTK-3[21] version of the WebKit library, which is installed by default[22]. Consequently, the additional installation of the old GTK-2 version, contained in the package `libwebkitgtk`, fixes the problem[23].
Furthermore, I reliably experienced crashes while using OpenJDK after closing dialogue windows in the browser prototype. The official Oracle JDK does not exhibit these problems. There are also known problems with the SWT browser in Eclipse 3.7 and 3.8 (they are fixed in version 4.0)[24].
The important thing to note here is that the support for the SWT browser is not given, not even for Eclipse. In most cases this should be fixable by installing one distribution package or by an upgrade of Eclipse. Nevertheless, it cannot be guaranteed that all current Saros users will be able to use the SWT browser.
Thus, a decision had to be made whether this was acceptable or not. On Monday 17 November 2014 Franz Zieris, Holger Schmeisky, Matthias Bohnstedt, and I decided that it was and the web-based approach could be pursued further.

Cross browser

As the SWT browser will use a different browser on every major operating system by default – Internet Explorer under Windows, Chromium under GNU/Linux, and Safari under Mac OS X – the application has to deal with rendering issues and differences between these browsers. The biggest downside is that the use of old Internet Explorer versions (prior to 9 or 10), which lack support of standards[25], cannot really be prevented because Saros still supports Eclipse 3.6 and the SWT version determines the version of the Internet Explorer to be used. However, the Javascript community has years of experience with such issues and libraries like JQuery[26] can help, too.

[20] `https://confluence.jetbrains.com/display/IDEADEV/IDEA+14.1+EAP` (retrieved 20 March 2015)
[21] SWT beginning with the current version 4.4 uses GTK-3 by default.
[22] There is also the remote possibility to try and use the Mozilla browser if WebKit does not work. Native Swing (see subsection 2.2.6) may be of help here.
[23] See `http://www.eclipse.org/swt/faq.php#browserlinux` (retrieved 20 March 2015) for detailed information about SWT browser compatibility under GNU/Linux.
[24] `https://bugs.eclipse.org/bugs/show_bug.cgi?id=404776` (retrieved 20 March 2015)
[25] I describe concrete problems of this kind in section 4.2.2
[26] `https://jquery.com/` (retrieved 20 March 2015)

License problematic

As the license of SWT (EPL) is incompatible with the GPL license of Saros, we will split the IntelliJ IDEA part into two plug-ins – one containing the SWT libraries, which will be licensed under Apache 2.0, and the other containing Saros/I, which remains under GPL.
I will now summarise the pros and cons of the use of the SWT browser for Saros.

Advantages

- Identical browser technology on the code level in IntelliJ IDEA and Eclipse.
- Possibility to use Björn Kahlert's improvement of the SWT browser.
- Embedding SWT into Swing produces interesting knowledge as it is generally discarded and discouraged without giving fundamental reasons[27]. There are numerous examples on how to embed the SWT browser in a Swing frame[28], but to the best of my knowledge there are no available implementations of doing this in an integrated fashion in a plug-in inside an application[29].

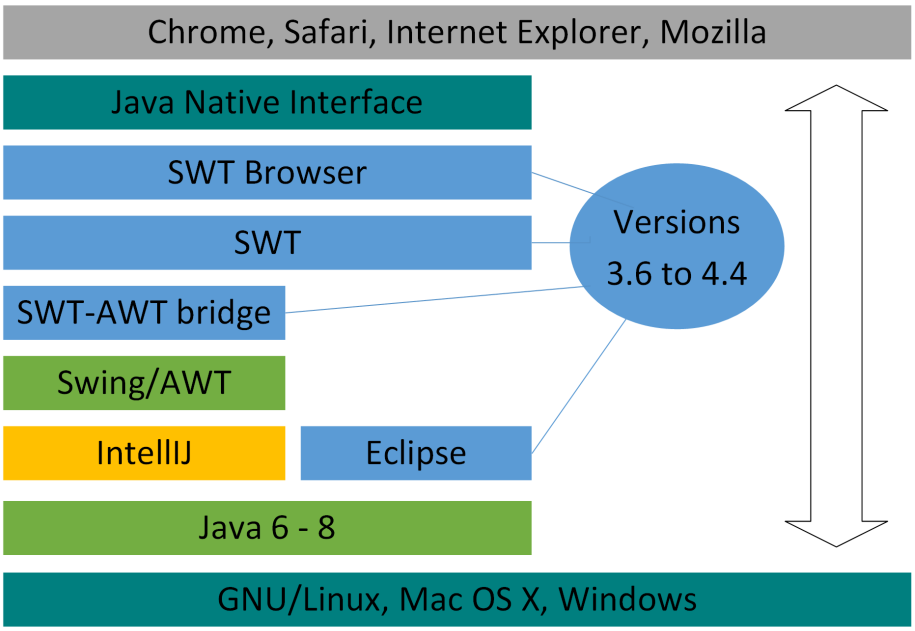

Figure 2.2: The technology stack for using the SWT browser

[27] The incorrect rendering when mixing Swing and AWT is gone with Java 6 Update 10 (see section 1.5)
[28] In fact, this is one of the most prominent examples why people do so in the first place as Swing does not include its own browser.
[29] The unofficial Vaadin UI Designer uses Native Swing (see section 1.6).

Disadvantages

- Compatibility and stability:
  - Requires a compatible browser to be installed and configured in the operating system (see Figure 2.2 for an overview of the technologies that have to play together and the supported range of versions.).
  - Further depends on the SWT version (SWT-AWT bridge implementation) and the Java version.
  - Known problems under Mac OS X and Linux.
- Uses Java Native Interface (JNI) which makes debugging difficult[30].
- Cross-platform issues inside the browser.

2.2.3 Björn Kahlert's improvements to the SWT browser

As mentioned in section 1.6, Björn Kahlert wrote a wrapper for the SWT browser. Part of my evaluation was to learn more about this wrapper and what services it can provide for the Saros plug-in.
Björn Kahlert presented his motivation for extending the SWT browser in an email to the Saros mailing list[31]. I will sum up the reasons that are relevant in this context here. The SWT browser:

- does not correctly detect when a website is loaded.
- does not calculate its own size correctly.
- might encounter security restrictions on some native browsers, e. g. Chrome when executing external Javascript code.
- provides no information about mouse coordinates, focused or hovered elements.
- does not synchronise the website's background colour with the default one for SWT widgets.

Björn Kahlert's component circumvents all of the issues above and furthermore

- allows methods to be executed from any thread by returning `Future` objects.
- provides converter methods for Javascript return values.
- improves the notification about Javascript exceptions significantly.
- can delay the execution of Javascript until the website is loaded.
- can inject Javascript libraries like JQuery so that they can be used independently of the website.

Features needed by Saros

A considerable amount of the component's methods and functionality does not seem to be needed by Saros (at least at the moment), such as the listening for Javascript events or managing scrolling positions. Others, like the thread-independent execution or Javascript exception notification, may be convenient for development but are

[30] During my thesis I myself experienced reproducible JVM crashes (segmentation faults) in two different situations (section 3.1 and section 4.2.2).
[31] `http://sourceforge.net/p/dpp/mailman/message/32780848/` (retrieved 20 March 2015)

not essential for a prototype. The prototype implementation revealed that the only functionality definitely needed by Saros is the ability to detect when the site is fully loaded, i. e. Javascript objects and methods are present, and delay Javascript commands up to this point. Without this functionality, Javascript functions might be executed before the corresponding objects have been initialised and would therefore be unreliable.

New development vs. extension

Due to extensive asynchronous method calls, which in turn might cause further asynchronous execution, the control flow inside the browser wrapper is hard to comprehend at first, so that it is not obvious how the changes needed to get it running on Windows and GNU/Linux should be applied (see subsection 4.2.2). Since the delayed Javascript execution is the only required feature, the alternative of implementing a new wrapper and copying functionality as needed has to be considered.
I made a sketch of a minimal re-implementation, which basically meant to delay the execution of methods until a condition is met. As the event dispatch thread must not be blocked while waiting, this results in asynchronous calls and dealing with the result. Section 4.2.3 explains the enclosed difficulties. My minimal sketch soon resembled the structure of Björn Kahlert's browser as this is exactly the problem it was trying to solve. Since a new implementation is expected to have a similar level of complexity and the unused methods should not hurt the functionality, the preferred choice is to adapt the browser. This also avoids re-inventing the wheel because even if the browser is used as a guideline, it is likely that some of the already existing functionality is implemented again.

How to use and extend it

The first choice would be to use Björn Kahlert's wrapper as it is. Unfortunately, it only works on Mac OS without modifications and contains Eclipse-specific code, which has to be deactivated for Saros.
Because of the necessary removal of Eclipse-specific code not all modifications can be included in the original repository. The next best option would be to fork the project (on GitHub) and try to include as many of the adaptations in the upstream repository as possible. However, the original project has a different aim. It provides a rich set of Eclipse widgets and utility classes – the browser component is only one of them – whereas the browser for Saros has to be independent of concrete GUI technologies.
To keep the browser fork simple, the browser widget is extracted and all other unused widgets and unneeded utility classes are removed. This results in a considerably different package structure such that changes cannot be integrated easily into the original. However, I tried to make each modification reproducible by using dedicated commits (see subsection 4.2.1). Since both projects have different aims and requirements, the focus lies on an unhindered adaptation for Saros. The hope was that over time, a general-purpose browser library would emerge, which in turn could theoretically be

used by Björn Kahlert's widgets.
The decision described above was made, after repeated conversation with Björn Kahlert, in the research seminar of the Software Engineering working group. The browser component is being developed separately from Saros but closely connected and driven by its needs. The main reason for the separate development is that the component may be re-used by other projects and that this prevents Saros' business logic from melding into the component.
For the evaluation decision I will not consider the browser wrapper separately from the SWT browser because at least some of its functionality is required. If the SWT browser is chosen, this functionality will be used as well – in one form or another. If the decision favours another browser than the SWT one, Björn Kahlert's browser must be either re-written to wrap another internal browser or the required functionality must be added to the other browser separately.

2.2.4 JxBrowser

The JxBrowser[32] (developed by TeamDev) is written in Swing and uses Chromium binaries for rendering. Although it is a commercial product, Saros as a free software project may use it free of charge, which Holger Schmeisky could confirm via direct email communication. If the `license.jar` is loaded in the classpath it takes no special steps to get it running inside IntelliJ IDEA. In theory, it is possible to embed it inside Eclipse via the SWT-AWT bridge, but it is not officially supported[33].
The JxBrowser has more features than Saros could possibly need, such as support for Flash or Silverlight. In addition, it provides some of the functionality of Björn Kahlert's browser, for example mouse listeners, but has no support for the required delayed Javascript execution. Interface-wise it is very similar to the SWT browser, so they might be interchangeable with reasonable effort (see section 4.2.7).

Advantages

- Same browser engine on every platform, i. e. no cross-browser issues and no Internet Explorer.
- Swing browser and thus no reliance on the SWT-AWT bridge for IntelliJ.
- No known compatibility issues.

Disadvantages

- Big footprint, about 200 MB, as it ships with Chromium binaries.
- Proprietary, the source code is unavailable.
- IntelliJ-only solution (or again use the SWT-AWT bridge).
- Special effort required to implement the delayed Javascript execution.

[32] `http://www.teamdev.com/jxbrowser`
[33] See this post by a TeamDev employee `https://groups.google.com/a/teamdev.com/forum/#!topic/jxbrowser-forum/twlTghkf1AA` (retrieved 20 March 2015).

2.2.5 JavaFX browser

JavaFX[34] is part of the Java platform as of Java 7 Update 6. It contains a full-featured, modern browser engine based on WebKit[35] and offers support for new standards such as HTML5.
As it requires no native browser and does not rely on a seldom-used third-party library like the SWT-AWT bridge, it is expected to run stably across all platforms. In this context, stable refers not only to runtime stability, but also includes compatibility across version updates. JavaFX reduces the number of key components affected by updates by two because neither the installed browser nor the SWT version matter, only the Java version.
The main drawback is that it requires at least Java 7. In section 3.2 I describe my unsuccessful effort to get it running under Java 6.
As IntelliJ IDEA on Mac OS X (see section 2.2.2) and Saros itself still require Java 6, the JavaFX browser has to be dismissed for now[36].

Advantages

- Same browser engine on every platform, no cross-browser issues.
- Advanced and much more comfortable data transfer between Java and Javascript (see section 4.2.7).
- Exclusive reliance on Java core functionality (as opposed to the SWT-AWT bridge).
- Has Oracle's support in the future and might replace Swing and possibly SWT.

Disadvantages

- Requires at least Java 7 (Java 6 and additional software under Windows), sensible to require Java 8.
- Relatively new and therefore few experience in contrast to SWT or Swing.
- At least the delayed Javascript execution has to be added.

2.2.6 DJ Native Swing

The Native Swing project[37] aims to provide an easy integration of some native components into Swing applications. It features a browser component that wraps the SWT browser and makes it available for Swing. It has been developed by a single developer who is still active but there have been no commits since July 2014.

[34] See subsection 1.5 for more information.
[35] A mature and free web browser engine used by Safari for example (`https://www.webkit.org/`, retrieved 20 March 2015).
[36] I wrote an email to the Saros mailing list about an upgrade to Java 7 for JavaFX but got neither support nor rejection at the time (`https://www.mail-archive.com/dpp-devel%40lists.sourceforge.net/msg01125.html`, retrieved 20 March 2015).
[37] `http://djproject.sourceforge.net/ns` (retrieved 20 March 2015)

It does much more than simply establish the connection between AWT and SWT. For example, it fixes the "illegal thread exception" under Mac OS (see subsection 3.1.3) and enables the use of Firefox under GNU/Linux, which is not trivial as SWT will probably not find the installed version on most distributions[38].

Unfortunately, the improvements around the browser are not isolated but integrated in a whole native interface, which must be initialised before the browser can be used. For example, Javascript commands are translated into general-purpose messages that are sent to the native interface and subsequently dispatched to the browser. The browser interface is slightly extended compared to the SWT browser, e. g. it provides more useful information about occurred exceptions, just like Björn Kahlert's wrapper. However, it lacks the ability to delay the execution of methods until a defined state is reached.

Due to the complexity of the source code and the required orientation I estimate the work effort of immediately combining this wrapper with the features from Björn Kahlert's components to be too high, since I already need to familiarise myself with Björn Kahlert's code. Furthermore, I am currently not aware of other problems that Native Swing can solve for Saros.

Since it is also a wrapper for the SWT browser, I will not view it as a separate option but as a pool of knowledge which might be useful if further compatibility issues arise.

Advantages

- Contains a lot of knowledge about the SWT-AWT connection.
- LGPL-License.

Disadvantages

- Fairly complex source code.
- Browser not isolated.
- No delayed method execution.

2.2.7 Summary

Since the JavaFX browser has to be dismissed for now, there are two main candidates, the SWT browser and the JxBrowser. Both introduced wrappers of the SWT browser are not viewed as a separate option for the next section. If the choice falls on the SWT browser, the features of the wrappers can be added when they are needed. Such a gradual process eases the learning curve for both Native Swing and the browser wrapper.

[38] See also `https://www.eclipse.org/swt/faq.php#howdetectmozilla` (retrieved 20 March 2015)

2.3 Criteria

I have devised the following criteria to evaluate the GUI technology used. Not all of these will be extensively covered in my thesis as I have just written the first prototype. However, I will consider them to detect possible problems and make a first assessment. Criteria:

- **Initial development effort** denotes the required effort to implement the Saros GUI for IntelliJ. This includes evaluation and initial orientation.
- **Ongoing development effort and maintenance**: The reduction of duplicate code and the complexity of the code base are the main issues I will look at. Where can duplication be avoided, where is this not possible, and where might there even arise new redundancy?
- **Presentation issues**: I will mainly check whether all of the current design elements can be displayed in the browser.
- **Response times and performance**: Are there any risks or opportunities regarding the user experience?
- **Stability**, especially support on the different operating systems and across Java versions
- **Ease of developing GUI tests**: the avoidance of platform-specific test code and the availability of frameworks are relevant here.
- **License**: Since all of the candidates can be used by Saros (even the proprietary JxBrowser), I will not elaborate further on this topic but it has been an important criterion during the evaluation.

2.4 Evaluation of an HTML-GUI vs. Java toolkits

In this section I will compare the browser-based approach with the conventional one based on the previously listed criteria using the properties of the browser components learned so far. In the following, each subsection is dedicated to one criterion.

Size of Eclipse's UI package First, I will give a few numbers to show that it makes sense to talk about the reduction of the code base.
I measured the size of the `ui` package of Saros/E using IntelliJ's statistic plug-in[39]. All numbers refer only to the Java code on Saros' master branch on 25 October 2014. Saros Eclipse's package `de.fu_berlin.inf.dpp.ui` contains 189 classes, 17,774 lines of real code and 5,578 lines of comments. It therefore comprises almost half of the entire Saros/E project which has 36,319 and 13,269 lines, respectively. By comparison, the common `core` package contains 20,905 and 10,470 lines. The current IntelliJ GUI implementation is not used as reference because it is incomplete and in large parts a copy of the Eclipse code.
The only thing important those numbers tell us is that the current GUI implementation is so big that it makes sense to think about reducing its size. Of course, this

[39] `https://plugins.jetbrains.com/plugin/?idea&id=4509` (retrieved 20 March 2015)

does not mean that an IntelliJ implementation will result in about 28000 redundant lines of code as most of the logic is IDE-independent. It is just wrapped inside SWT classes, e. g. classes that extend `Composite` or `Action`. The current code is so large because it mixes presentation logic, presentation and domain data, and framework specific classes.

2.4.1 Criterion: Initial development effort

The browser-based GUI code is expected to be leaner since the GUI is described declaratively via HTML and the design is done separately via CSS. This declarative GUI will eliminate the need for SWT's nested containers or hierarchical GUI classes. Looking at the current Eclipse implementation, there are a number of GUI model classes like `RosterContentProvider` that encapsulate domain values and their corresponding presentation. In HTML, there will still be GUI model classes, but those classes can be implemented agnostic of how their content gets displayed.
Parts of the Eclipse GUI code can be abstracted – mainly because they contain business logic – and be re-used for IntelliJ so that the size of the duplicated GUI code is reduced. However, this introduces overhead as a result of the additional abstractions and requires an effort which may be comparable to writing a new GUI.

Browser

+ Modern web technologies are available.
+ Declarative GUI results in less and more readable code.
– Unknown terrain: missing knowledge especially for the browser integration and the connection between Java and Javascript.
– Additional effort required to integrate the browser.

Swing and SWT

+ Linear development, analogous to Saros/E.
+ No additional browser layer.
– A new GUI for Saros/I has to be implemented anyway.
– Refactoring of Saros/E required to remove unnecessary coupling of logic in GUI classes.

Both alternatives require the creation of a sizeable GUI as well as a refactoring of the Eclipse GUI code. Due to the additional browser layer, the HTML-based approach will probably require more effort and familiarisation. However, in the process interesting knowledge will be gained about this new approach to writing IDE plug-ins and the embedding of a browser component.

2.4.2 Criterion: Ongoing development effort

I will characterise this mainly by the amount of redundant code and the size of the code base in general. Another essential factor is the complexity and readability of the

code. As pointed out in the previous section, the HTML GUI is expected to reflect the visible structure more clearly in the code and to separate styling and logic. The lack of Java class hierarchies for GUI elements reduces complexity.
Real GUI code that has to be duplicated in the traditional approach regardless of how many abstractions are introduced and could be unified in the HTML version includes the Saros view with roster, chat, and menu bar as well as about ten wizards with one to four pages. Even though this is a manageable amount, the lines of code required to express this in Java are unproportionally many.

Browser

+ Less duplication and less code altogether.
+ Plain, structure revealing GUI code and no need for Java GUI classes.
+ Easier to apply changes to the style.
− Mix of technologies (Java and Javascript).
− Boilerplate code for communication and data exchange between Java and Javascript.

SWT and Swing

− Duplication of the GUI views and elements.
− Old technologies and verbose Java GUI toolkits.
− Code will probably contain more redundancy than necessary as there is no clear separation between logic and presentation.

The main potential of the HTML-based solution lies in the reduced code size and complexity. The reduced maintenance effort might soon amortise the higher initial development costs.

2.4.3 Criterion: Presentation problems

During the evaluation I wrote HTML sample code for all different types of existing Saros GUI elements:

- The contact list and the session tree: both are (nested) lists with similar structure, so I implemented only the contact list, but considered that the list may contain differently styled text parts and images.
- The context menu: the default right-click menu of the browser had to be deactivated and replaced with a context sensitive one.
- The drop-down menu and menu bar: I focused on the drop-down menu as it is the most complicated element of the menu bar.
- The dialogue wizards: again I implemented a scenario that is probably more complicated than necessary. There may be multiple simultaneous dialogues that are implemented as external dialogue windows displaying a separate browser. In practice it might suffice to use one pop-up window or display the wizards inside the Saros view altogether. It may also be sufficient to allow just one dialogue at a time.
- A progress bar for long-running tasks.

I noticed the following limitations. In the Eclipse version the menu bar is part of the view's frame, i. e. it is outside of the actual display area of the browser. This would have to change, but is a minor cosmetic issue and should barely be noticed by the users. Alternatively, the menu-bar could be implemented in an IDE-specific fashion. Resizing the browser's content when the size of the view changes requires additional effort, but I was able to solve it even without the help of Björn Kahlert's wrapper.
The following two minor problems are fixable as well. First, the context menu has to be rendered upwards when activated on the bottom of the screen[40].
Second, I noticed that the HTML component did not gain focus every time the mouse pointer entered the component. As a consequence, buttons had to be double-clicked because the first click only gave the focus to the component.

Browser

+ Huge range of options, whole palette of browser capabilities.
+ Every individual pixel can easily be changed.
− Limited to the browser area.
− Integration into the look-and-feel requires additional steps.

SWT and Swing

+ Excellent IDE integration.
− Limited set of widgets.

2.4.4 Criterion: Performance

I did not notice any performance problems, but this is to be expected in the current prototype of the user interface with next to no elements.
A performance critical situation might arise when opening the view as the browser is initialised and the entire website is loaded. However, this is currently only a theoretical problem.
The rendering performance of SWT is expected to be inferior to that of web browsers. Björn Kahlert experienced this when he was adding many images to his widgets, which was not practical in SWT. Performance might also be an issue for Saros as the activity log currently has significant performance problems when adding hundreds of lines (see [Dur14]).

2.4.5 Criterion: Compatibility and stability

This is the biggest problem of the browser approach. While the native SWT and Swing solutions bear no risks, depending on the browser technology I could identify considerable limits. JavaFX requires a Java version greater than 6 and the SWT browser has known issues on GNU/Linux and is incompatible with Java 7 and 8 in

[40] In the current Saros version the context menu will only appear on the bottom if the view is very small.

IntelliJ on Mac OS (see section 2.2.2). The JxBrowser also depends on the SWT-AWT bridge when used inside Eclipse and therefore is affected by the Java 7 and 8 issues on Mac OS.
Although this seems manageable with small limitations, the real impacts remain to be seen as the prototype matures.

2.4.6 Criterion: Testability

Unfortunately, an HTML page in an embedded browser cannot be tested using standard web testing and automation frameworks like Selenium (webdriver)[41], Watir[42] or Sahi[43] as those need to render the website themselves (see subsection 4.3.6). However, identifying elements and triggering events for them is easy in HTML, so that the needed functionality can be implemented in Javascript even without a framework. The IntelliJ platform lacks a GUI test framework or a GUI automation tool. There are two automation tools available for Swing UIs, but I only managed to implement a basic task, like opening a Tool Window in one of them; even this required extensive trial-and-error programming (see section 4.3)

Browser

+ Much easier to implement.
− No frameworks.
− Cross-browser problematic.

SWT and Swing

+ Eclipse side is already implemented.
− Swing automation frameworks are difficult to use and do not work properly in all circumstances.

2.4.7 Decision for browser

Figure 2.3 sums up the comparison for each criterion. The Criteria, performance, and presentation issues are left out as they don't currently favour one side or the other.
The greater initial development cost can be justified by the expected return of investment in terms of lower future development costs. Compatibility is a problem but should be manageable. For example, the browser component could be changed after support for Java 6 has been dropped or it is even possible to use different browsers for different constellations.
To summarise, I think it is quite possible to display the Saros plug-in in a web browser in both IntelliJ and Eclipse with two caveats. There is a small percentage of users (especially on GNU/Linux) that will have to execute manual steps (install packages

[41] `http://www.seleniumhq.org/` (retrieved 20 March 2015)
[42] `http://watir.com/` (retrieved 20 March 2015)
[43] `http://sahipro.com/` (retrieved 20 March 2015)

Criterion	Browser	SWT and Swing
Initial development effort	–	0
Ongoing development effort	++	–
Code duplication	++	–
Stability	–	++
GUI tests	+	–

Figure 2.3: Tabular summary of the comparison

or upgrade their Eclipse installation) to get the plug-in running. The choice of the used browser component may change over time, as JavaFX is not yet an option and the SWT browser may lose runtime support in the future. This will be elaborated upon in the next section, when the choice is actually made.

2.4.8 Possible browser components

The following two browser candidates remain:

- SWT browser (Eclipse and IntelliJ).
- JxBrowser (IntelliJ and possibly Eclipse).

There are different possible constellations for running a web-based Saros plug-in because Eclipse and IntelliJ may use different browser components. Using different browsers has the decisive disadvantage that the code for adding the required functionality to the browser is duplicated.
I compared JxBrowser and SWT Browser for usage inside IntelliJ, which is illustrated in Figure 2.4.

Criterion	SWT browser	JxBrowser
Development effort	manageable but at least somewhat hacky	minimal effort for integration, special effort for extension of browser
Code duplication	same browser code	duplicated code
Stability	best effort, may be sufficient	expected to be stable
Cross-browser	requires special effort	only one browser on all systems
Size	ca. 20 MB	ca. 200 MB
License	free	free of charge, proprietary

Figure 2.4: Tabular comparison of SWT browser and JxBrowser

The footprint of the JxBrowser is difficult to justify and the duplicate code around the browser makes it just a fallback option for Mac OS, for example.

2.4.9 Possible combinations

- SWT browser in both IDEs: same browser code but IntelliJ side less robust
- SWT browser in Eclipse and JxBrowser in IntelliJ: duplicate browser code but more robust
- JxBrowser on both sides: discarded because in a one-on-one comparison the SWT browser wins

On the IntelliJ side there are basically two alternatives: the embedding of either the SWT browser or the JxBrowser.
Since the JxBrowser has considerable disadvantages and the SWT browser is the only sensible option for Eclipse anyway, it is the first logical step to try the same set-up for IntelliJ as well. If severe problems occur, the usage of an extra browser for IntelliJ can be the second step.

2.5 Side note: alternatives

2.5.1 Using a browser only for IntelliJ

This would keep the Eclipse side untouched because the browser is used for the new IntelliJ implementation only. It might be sensible to gradually introduce the new technology. However, one of the positive side effects of integrating the browser in both IDEs is that the Eclipse GUI code gets refactored along the way. Furthermore, it might be hard to include Eclipse later on as its specific requirements have not been taken into consideration from the beginning.

2.5.2 IntelliJ implementation completely in SWT

There are two problems with re-using the current Eclipse GUI, mostly as it is, for IntelliJ.
First, implementing the IntelliJ GUI entirely in SWT would have been worth a try if the current Saros/E implementation was not so tightly coupled to Eclipse. It not only uses SWT code but also heavily relies on the JFace library and the Eclipse platform, e. g. the menu bar is an Eclipse element. The introduction of abstractions would be too invasive to have the benefit of not having to write new code.
The second problem is posed by the compatibility issues of the SWT-AWT bridge on Mac OS X with Java 7 and 8 (see section 2.2.2).

2.5.3 Native JavaFX implementation

If JavaFX runtime support was not an issue, this alternative would have to be seriously considered because this technology has the potential to replace SWT and Swing and features a declarative GUI as well as seamless embedding of web technologies. Integrating it in SWT and Swing applications seems to be no problem.

2.6 Summary

The decision was made to further pursue the browser-based GUI as the evaluation so far has identified a promising browser candidate and has illustrated the possible advantages compared to a separate SWT and Swing development. However, it is important to emphasise that this is only the first part of the evaluation. The next step is to develop the prototype further and have a look at the resulting code to see whether the estimates for development effort and code duplication are correct.
Furthermore, the decision for the use of SWT browser implementation is not final. The main problem is that JavaFX would be the clear choice in maybe a year from now but currently does not meet the runtime requirements for Saros. The SWT browser's support in the future is uncertain, at least for Mac OS. Therefore, I will keep in mind that the browser might change while I am implementing the prototype.

3 Details of the evaluation phase

This chapter describes the implementation necessary to embed the SWT browser in IntelliJ on the different operating systems (section 3.1). It further contains research about the requirements of JavaFX (section 3.2) and my assessment of the usage of AngularJS for Saros' HTML part (section 3.3).

3.1 Using the SWT Browser in IntelliJ

Embedding the SWT browser in IntelliJ consists of two parts.
Firstly, the SWT browser is made available in Swing via the `SWT_AWT` class that is included in SWT for the purpose of using SWT and Swing (or AWT) components in each other's toolkit. Although Swing's widgets are independent of AWT, they still need to be endorsed in native AWT containers[1]. Secondly, the SWT browser has to be integrated in a plug-in and in the user interface of the IntelliJ. For the first part there are a few examples to be found on the internet. Gordon Hirsch's article[2] provides a detailed overview on that topic along with useful background information. However, there are no examples for doing this inside a plug-in, which poses additional challenges that I will explain now.

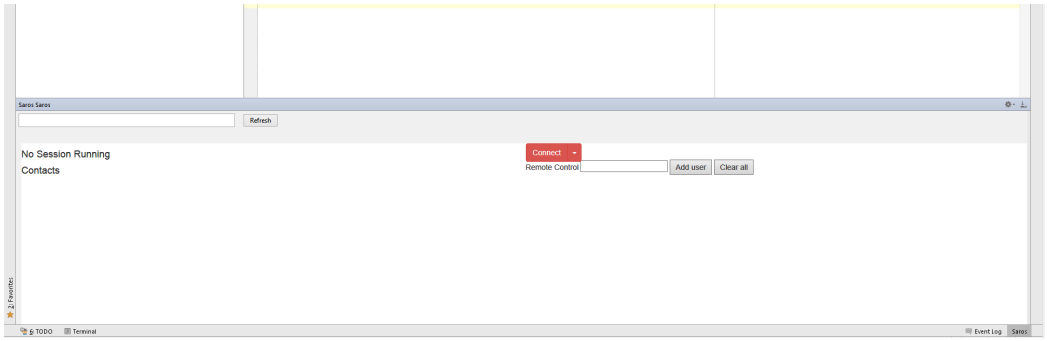

Figure 3.1: A Tool Window in IntelliJ

[1] Compare `http://stackoverflow.com/a/408830` (retrieved 20 March 2015)
[2] `http://www.eclipse.org/articles/article.php?file=Article-Swing-SWT-Integration/index.html` (retrieved 11 April 2015). As the article is from 2007, some of the details may be outdated, e. g. the compatibility issues section. See the SWT FAQ under `http://www.eclipse.org/swt/faq.php` (retrieved 20 March 2015) for updated information.

IntelliJ plug-ins use so-called Tool Windows to display their user interface[3]. These Tool Windows are integrated into the IntelliJ main window and look like the one in Figure 3.1, which displays an early HTML prototype of Saros. Their main feature is that they can be hidden and shown again.

Displaying the browser in IntelliJ for the first time

On the code level, the Tool Windows display an instance of Swing's `JPanel`; Saros' derived class for the browser is called `SwtBrowserPanel`. This class contains the `SwtBrowserCanvas`, an AWT component, which is used as a parent for the subsequently created SWT `Shell`. The creation process that is started when the user opens the Tool Window is illustrated in Figure 3.2.
One limitation of the `SWT_AWT` class is that the AWT `Canvas` object must be visible before the SWT `Shell` can be created. The reason for this is that when creating the SWT `Shell`, a native `getHandle()` call is made to get a representation of the AWT `Canvas` in which the `Shell` should be embedded. This call fails unless the component is visible. In a standalone Swing application it is sufficient to place the `new_Shell()` call after `frame.setVisible(true)`. Inside IntelliJ the equivalent call that makes the `Canvas` visible is `contentManager.addContent(content)` which is called in the factory method `createToolWindowContent()`. As a result, the embedding code had to be split into the creation of an empty panel and the initialisation of the browser that is triggered afterwards. The example code from Caprica[4] which I used as basis for the SWT embedding had to be transformed to work inside a `JPanel` instead of a `JFrame` and extended by a call to `shell.setSize(int width, int height)` in order to actually display the browser. Otherwise the Tool Window silently stays grey.

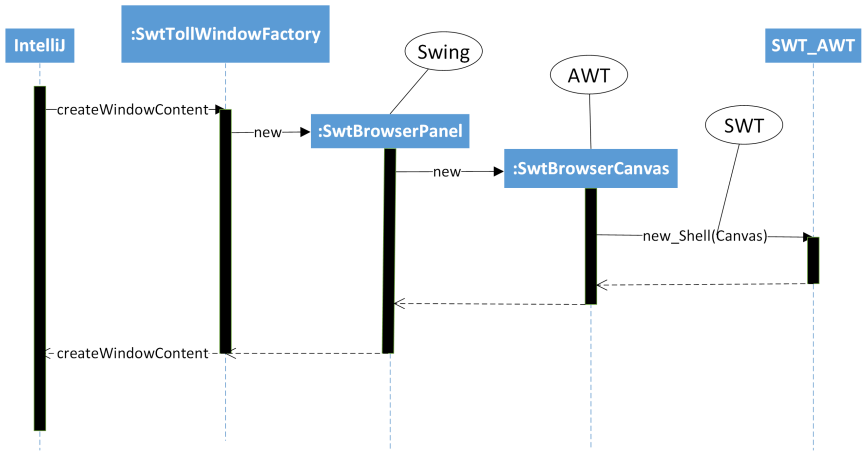

Figure 3.2: Sequence diagram of the establishment of the SWT AWT bridge

[3]The Vaadin UI Designer (see section 1.6) uses an editor window to display its browser because it provides an editor instead of a control panel.
[4]`https://gist.github.com/caprica/6890618` (retrieved 20 March 2015)

3.1.1 JVM crash when closing the browser

The browser is now displayed inside the Tool Window. Here lies the difference to the single frame application Caprica's example was written for; the Tool Window constantly gets hidden and re-shown. In the example code, when the frame is closed, all SWT components are disposed of and the JVM terminates, whereas inside the Tool Window the JVM does not exit and the SWT components may have to be re-created. The example code therefore calls `Display.dispose()` after its SWT event dispatch loop ends. Making this call inside the IntelliJ plug-in crashes the JVM under Windows. As it does not crash on GNU/Linux, this may point to a bug in the Windows implementation of SWT.
I proposed two approaches for dealing with this problem:

1. Finding a way to dispose the `Display` cleanly when the window is hidden and create a new one when the window is visible again. This basically requires the debugging and fixing of the JVM crash or applying a work-around that does not trigger it. Matthias Bohnstedt and I did not manage to do this in our self-imposed timeframe of three working days. Additionally, I found the requirement to display two browsers simultaneously if dialogue wizards should be used. This approach does not work properly with multiple browsers as the number of existing displays is very limited. According to the Javadoc of the `Display` class, "some platforms which SWT supports will not allow more than one active display"[5].
2. Not disposing the display but keeping it fully operational after the Tool Window is closed. Based on this idea I extended the event dispatch method so that it does not end but holds on to the Display. This works, but the resulting code in Listing 3.1 is rather complex as it contains nested loops and duplicates the event dispatch logic. The SWT `Shell`, which is more or less a sub element of the `Display` (there can be multiple `Shells` per `Display`), and the associated browser get disposed automatically when the Tool Window is closed. I found no way to circumvent this, but it is not a problem as new instances in the same `Display` can be created as long as the `Display` is in a consistent state and not disposed.

```
public void run() {
    Display display = new Display();
    while (!display.isDisposed()) {
        [...]
        Shell shell = SWT_AWT.new_Shell(display,
            SwtBrowserReuseDisplayCanvas.this);
        final Browser browser = new Browser(shell, SWT.NONE);
        [...]
        shell.open();
```

[5] I wrote a small class which tests how many displays can be created before a "no more handles" `SWTException` is thrown. On my Windows machine this number was 32.

```
        // Execute the SWT event dispatch loop...
        while (!isInterrupted() && !shell.isDisposed()) {
            if (!display.readAndDispatch()) {
                display.sleep();
            }
        }
        reopened = new AtomicBoolean(false);

        // second event dispatch loop: not ideal
        while (!reopened.get()) {
            if (!display.readAndDispatch()) {
                display.sleep();
            }
        }
    }
}
```

Listing 3.1: Naive implementation of the persistent display

Furthermore, this code cannot be reused to create a second shell and browser for dialogue windows because the creation logic is intertwined with the event dispatch loop. After I carried out multiple refactorings and separated thread management and browser creation concerns, I ended up with the simple code of Listing 3.2. I also replaced `new Display()` with the preferable `Display.getDefault()` that only creates a new Display if it cannot find an existing one.

```
class SwtThread extends Thread {
    [...]
    public void run() {
        Display display = Display.getDefault();
        // Execute the SWT event dispatch loop...
        while (!display.isDisposed()) {
            if (!display.readAndDispatch()) {
                display.sleep();
            }
        }
    }
}
```

Listing 3.2: The SWT event dispatch thread

In hindsight, it would probably have been easier to rewrite this part from scratch instead of extending the example as the coupling of the `Shell` creation to the event dispatch thread (EDT) made me think in too complex a fashion. As I learned later, the `SwtThread` above contains the standard EDT loop.

Since the `Shell` and the SWT browser get disposed every time the Tool Window is closed, they have to be reactivated when the window becomes visible again. Finding the right listener which gets notified in this event proved to be difficult as the IntelliJ

plug-in documentation contained no information[6]. Adding a `StateChangeListener` to the `ToolWindow` does not work properly as the listener does not contain the previous state and is triggered too often with the status "visible".
After browsing the source code of a number of other Tool Windows, I found an appropriate listener. `AncestorListeners` are informed when changes occur to one of the ancestors of a Swing component in the component hierarchy, for example when a component is added or made visible. The resulting call is shown in Listing 3.3.

```
addAncestorListener(new AncestorAdapter() {
    @Override
    public void ancestorAdded(AncestorEvent event) {
        createWebBrowser();
    }
});
```

Listing 3.3: Listening for the opening of the Tool Window

3.1.2 Dynamic loading of SWT library classes

As SWT's implementation is platform dependent, every operating system requires a different binary version. For that reason, the SWT library has to be loaded dynamically depending on the processor architecture and the operating system.
Matthias Bohnstedt wrote a first version of the `SwtLibLoader` based on this example[7], which only works outside of IntelliJ. To get it working inside IntelliJ, I had to adapt the code in two places. First, the `ClassLoader` has to be the `PluginClassLoader` of the Saros plug-in, which can be obtained as shown in Listing 3.4. Previously, the system class loader had been used.

```
ClassLoader classLoader = SwtLibLoader.class.getClassLoader();
if (!(classLoader instanceof PluginClassLoader)) {
    throw new RuntimeException("Unable to get hold of the plugin
        classloader");
}
PluginClassLoader pluginClassLoader = (PluginClassLoader)
    classLoader;
```

Listing 3.4: Usage of the appropriate class loader

Second, the plug-in directory containing the SWT library has to be found even if it differs from the default. Consequently, I had to get rid of the guessing of the path to the IntelliJ runtime folder. The IntelliJ plug-in API provides the method `PathManager.getPluginsPath()` which accomplishes this.
Furthermore, IntelliJ IDEA has to be prevented from automatically loading the library files because this would result in class name conflicts as the majority of classes

[6] As a side note, the IntelliJ source code is very sparsely documented, even public API methods needed for plug-in development.
[7] `https://www.chrisnewland.com/select-correct-swt-jar-for-your-os-and-jvm-at-runtime-191` (retrieved 20 March 2015)

are contained in each of the six bundled binaries. It is sufficient to not declare the folder as source folder or library and give it a name other than `lib` or `classes`. However, one small drawback is that the extra SWT library plug-in has to be deployed via a copy of the containing folder now because the run button in IntelliJ does not work for that anymore, due to the missing source and library folder definitions.

3.1.3 Getting the IntelliJ version to run on Mac OS

The implementation of the SWT event dispatch thread in Listing 3.2 throws an `SWTException` on Mac OS because the "Display must be created on main thread due to Cocoa restrictions".
The standard workaround is to pass `-XstartOnFirstThread` as JVM parameter at start-up. The Eclipse IDE does this for example. Due to the lack of alternatives I tried starting IntelliJ with this parameter but the error persisted – never mind that Saros is not able to do this in production anyway.
As Matthias Bohnstedt and I could find no other solutions on the Internet, I looked for alternative browsers. In this context I re-examined the browser from Native Swing (see subsection 2.2.6) which, in the end, is also a wrapper for the SWT browser and supports Mac OS X.
It was not obvious at first what it does to fix the problem, since it uses a lot of additional code around the browser and initialises a whole so-called native interface, in which the browser and other SWT components[8] are later created.
When I found out that the code in Listing 3.5 accomplishes the task, I searched for more information about the method used and this forum post[9] as well as the related bug tracker entry[10] confirmed the solution. It uses reflection to get the Mac-specific `MainQueueExecutor`, which runs on the main thread.

```
public static void runWithMacExecutor(final Runnable runnable) {
        [...]
        Object dispatch = Class.forName("com.apple.concurrent.
            Dispatch").getMethod("getInstance").invoke(null);
        Executor mainQueueExecutor = (Executor) dispatch.
            getClass().getMethod("getNonBlockingMainQueueExecutor
            ").invoke(dispatch);
        [...]
        mainQueueExecutor.execute(runnable);
}
```

Listing 3.5: Starting the SWT thread on Mac OS X (adapted from `SWTNativeInterface`'s `runWithMacExecutor` method)

I successfully tested my fix as described above under Mac OS 10.6 and 10.9.

[8] Native Swing also offers an HTML Editor, a Multimedia Player and a Syntax Highlighter (see `http://djproject.sourceforge.net/ns/`, retrieved 20 March 2015).
[9] `https://www.eclipse.org/forums/index.php/t/414910/` (retrieved 20 March 2015)
[10] `https://bugs.eclipse.org/bugs/show_bug.cgi?id=303869` (retrieved 20 March 2015)

After seeing the complexity and the amount of code Native Swing added around the SWT-AWT connection, I think it might be useful for other problems as well and should be considered a pool of knowledge for this topic (its forum and bug tracker as well).

3.2 Requirements of JavaFX

As Saros in general and IntelliJ IDEA on Mac OS must support running under Java 6 (section 2.2.2), I investigated whether it was possible to use JavaFX with Java 6. First, however, I will describe the general integration process on both IDEs (using Java 8).

3.2.1 Embedding the JavaFX browser into IntelliJ

JavaFX can easily be integrated into Swing applications[11] (and vice versa) and after minor adaptations of the standalone example[12] the browser is displayed inside the Tool Window.
The closing and re-opening of the Tool Window causes some problems (as it does with the SWT browser). After it has been closed, the browser does not start again because the JavaFX toolkit recognises that it is idle and shuts the JavaFX platform down. Unfortunately, the platform cannot be re-initialised in the same JVM. Setting `Platform.setImplicitExit(false)` prevents the platform from exiting in idle situations and solves the problem.

3.2.2 Embedding it into Eclipse

JavaFX can also be easily integrated into SWT applications[13]. Integrating it into Eclipse poses no additional difficulties as long as the required library bundles are found (see the next section).

3.2.3 Using JavaFX under Java 6

As the JavaFX runtime can be installed separately for Java 6 on Windows 32 and 64 bit (but only on Windows), I thought it might also be possible to distribute JavaFX with Saros if it is not installed.

[11] `http://docs.oracle.com/javafx/2/swing/swing-fx-interoperability.htm` (retrieved 20 March 2015)
[12] `http://docs.oracle.com/javafx/2/webview/jfxpub-webview.htm` (retrieved 20 March 2015)
[13] `http://docs.oracle.com/javafx/2/swt_interoperability/jfxpub-swt_interoperability.htm` (retrieved 20 March 2015)

JavaFX in Eclipse with Java 6

Eclipse loads used libraries from OSGI bundles. Unfortunately, I did not find a JavaFX bundle for Java 6. Efxclipse[14] which provides runtime support for JavaFX in an OSGI environment requires Java 8. I tried the older release 0.1.1 of Efxclipse which should enable running under Java 6[15] if compiled with Java 6 compliance level, but had no success.
Only when I included the JavaFX JAR file directly in the Saros plug-in could it be loaded inside Eclipse. However, if JavaFX is not installed on the machine it will still throw a runtime exception because it cannot locate the necessary DLL files. I tried the packaging tips from this Stack Overflow thread[16] and the JavaFX packager following the official documentation[17] with no success.
If the required DLL files are copied from a suitable Java 7 installation into the plug-in, JavaFX will start under Java 6 in Eclipse.
As IntelliJ uses no OSGI environment, including the JavaFX JAR is already the default procedure and with the additional DLL files it will run in IntelliJ with Java 6 under Windows, too. Out of curiosity I tried the same on GNU/Linux and it works if provided with the corresponding .so files (the GNU/Linux equivalent of DLL).
Nevertheless, it is not practical to copy compiled libraries from a different installation, distribute them with the plug-in and hope for the best because the files are specific to one operating system and architecture and this approach basically means distributing a part of the JRE and managing updates to it. Therefore, I had to discard the idea of using JavaFX with Java 6.
In the next section I investigate whether it is sufficient to require Java 7.

3.2.4 Using JavaFX under Java 7

As of Java 7 Update 6 JavaFX is part of the Java installation. Since Java 7 excludes the JavaFX JAR file from the classpath by default, it still has to be distributed. This is manageable, though, because the JAR file itself is independent of the operating system.
There are two catches:

- As already explained in the previous section, I did not find an OSGI bundle for Java version below 8 (though it is possible that I missed something) but the JAR file could still be included directly.
- OpenJDK does not include JavaFX. There is a separate OpenJFX project[18] that had no GNU/Linux installation packages at the beginning of my thesis (it

[14] `http://www.eclipse.org/efxclipse/index.html` (retrieved 20 March 2015)
[15] According to `http://tomsondev.bestsolution.at/2012/09/27/efxclipse-0-1-1-released/` (retrieved 20 March 2015)
[16] `http://stackoverflow.com/q/11349303` (retrieved 20 March 2015)
[17] `http://docs.oracle.com/javafx/2/deployment/packaging.htm` (retrieved 20 March 2015)
[18] `http://openjdk.java.net/projects/openjfx/` (retrieved 20 March 2015)

does now). The current OpenJFX version targets Java 8, but there is also a back-port for Java 7.

All in all, my impression is that Java 7 would be sufficient if one required the official Oracle version. Otherwise, it makes sense for Saros to skip Java 7 and switch to Java 8 straightaway, which many JavaFX related projects like Efxclipse do.

3.3 Web technologies

In this section I will shortly describe my thoughts and experiences on current web technologies. As this is not the main focus of my work, I only gathered information by implementing the necessary components for the HTML prototype. I made no decision about which Javascript frameworks should be used for future development.

3.3.1 AngularJS

First of all, I decided to write my prototype using AngularJS[19] for the following reasons.

AngularJS incorporates an MVC[20] architecture which leads to descriptive, intention-revealing HTML-code. Listing 3.6 shows exactly how the contacts are rendered.

```
<ul id="contact-list">
  <li ng-repeat="contact in contacts">
   <span>{{contact.name}}, {{contact.status}}</span>
  </li>
</ul>
```

Listing 3.6: A contact list in AngularJS

As opposed to a typical JQuery solution shown in Listing 3.7.

```
<ul id="contact-list">

</ul>
```

Listing 3.7: A contact list for JQuery

where the HTML code gets added dynamically via Javascript in Listing 3.8.

```
addContact = function(contact) {
    $('#contact-list').append("<li><span>" + contact.name + ", "
        + contact.status + "</span></li>");
}
```

Listing 3.8: Dynamically adding the contact using JQuery

[19] https://angularjs.org/ (retrieved 20 March 2015)

[20] Model-view-control see https://en.wikipedia.org/w/index.php?title=Model%E2%80%93view%E2%80%93controller&oldid=653645961 (retrieved 31 March 2015)

Here, domain code and presentation logic get intermixed in the controller, which reduces readability and can make changes harder because of the mix of different concerns.
I know that there are template engines which enable similarly descriptive presentation code but they complicate the prototype unnecessarily. Furthermore, I have used JQuery on multiple projects before and Björn Kahlert's browser component uses it extensively so I have a pretty good understanding of what the solution would look like in JQuery. On the other hand, I cannot say the same for AngularJS, so I wanted to see how easily AngularJS could be integrated and what its drawbacks are.

The drawbacks of AngularJS

AngularJS was written with typical websites in mind, where the application mostly reacts to users' requests made in the frontend. In Saros, a considerable amount of input comes from the network via the Saros core and nearly all of the user's input has to be processed by the Saros core. Consequently, the automatic synchronisation between the Javascript model and the rendered website in AngularJS is of little use to us. This feature is particularly convenient for user inputs which have direct implications for other elements on the same website. Additionally, Saros does not have a backend service interface from which AngularJS could require data via AJAX requests for example to fill its model. Even if this is added, some information still has to be pushed to the frontend, e. g. the appearance of a contact or an incoming message. Nevertheless, it is still possible to code the Saros GUI using AngularJS, but the main advantage we get from doing so is the nice and descriptive HTML code.
AngularJS' strict implementation of the MVC pattern hides the content of the model from the outside. As a result, additional JavaScript wrapper code has to be written for every method that should be callable from Java.

```
app.controller('ContactListCtrl', function ($scope) {
    $scope.contacts = [];

    $scope.add = function (contact) {
        $scope.contacts.push({name: contact, status: 'online',
            special: 'connecting'})
    };
});

__angular_addChatUser = function (username) {
    var exposedScope = angular.element(document.getElementById('
        contact-list')).scope();
    exposedScope.$apply(exposedScope.add(username));
};
```

Listing 3.9: Calling AngularJs methods from Java

This looks something like Listing 3.9. The first block defines the AngularJS controller with the model element `contacts` and an `add` function. The second block defines the

wrapper function which is callable from Java and has to acquire the scope variable. Bastian Sieker later used the event bus from the Backbone Javascript framework (`http://backbonejs.org/`, retrieved 20 March 2015) to execute methods inside the scope. The resulting code is much cleaner, but the complexity has increased due to the new event mechanism.

3.3.2 Libraries

I wanted to see whether more complex GUI elements like drop-down menus or a context menu pose any general implementation or presentation difficulties in Javascript. Therefore, I searched and found libraries to quickly develop proof-of-concept implementations.

UI Bootstrap

UI Bootstrap[21] by the AngularUI team is a collection of reusable and already styled AngularJS widgets. It is very easy to embed these in an existing AngularJS application as it has minimal dependencies. I used it to implement the connect button's drop-down menu.

Context menu

I used a very small angular context menu library[22] to replace the context menu. This library is easy to use and should meet our requirements; only the placement of the menu when activated on the bottom of the page might require small modifications. There are equivalent solutions for JQuery.

3.3.3 My assessment

The descriptive HTML is the main advantage of AngularJS compared to JQuery. The additional Javascript overhead of AngularJS might not be a concern as it can be abstracted. However, with my current knowledge I would suggest to prefer JQuery or another framework over AngularJS because Saros does not match the primary target group.

[21] `https://angular-ui.github.io/bootstrap/` (retrieved 20 March 2015)
[22] `https://github.com/ianwalter/ng-context-menu` (retrieved 20 March 2015)

4 Implementation

This chapter describes the implementation of the HTML GUI. It is partitioned into the implementation of the newly created `ui` module (section 4.1), which contains the actual GUI, the implementation of the used browser library, formerly known as Björn Kahlert's browser wrapper (section 4.2), and the implementation of the GUI test framework (section 4.3). In the last sections I will present refactorings I made along the way outside of the GUI (section 4.4) as well as the integration into the build process (section 4.5).

4.1 GUI module

For the implementation of the new HTML GUI a new module was created that can be seen as an extension of the `core` module. Both modules (`core` and `ui`) are bundles in Eclipse's OSGI context. Outside of an OSGI environment this additional configuration is ignored and causes no harm.

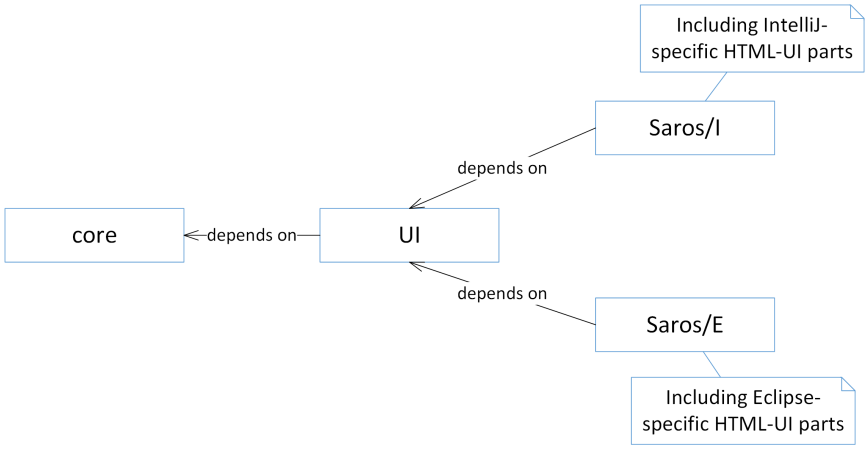

Figure 4.1: Dependencies between the individual Saros modules

The resulting dependencies between the Saros modules are displayed in Figure 4.1. Saros/E and Saros/I contain the code for the SWT and Swing implementations of Saros respectively. In addition, they now include the IDE-specific code for the HTML GUI. The `ui` module contains the IDE-independent part of the new HTML GUI. Only the embedding of the browser into the IDE (see chapter 3) and the creation of dialogue windows are IDE-specific. As the development progresses, the IDE-specific part will

also address further aspects, such as the different abstractions for workspaces and projects.
In addition, the `ui` module contains a resource folder with the HTML, CSS, and Javascript files that describe the actual design and structure of the user interface. The purpose of the accompanying Java classes is the data transfer between the Saros core and the GUI. Since the data transfer must work in both directions, there are Java methods, in the following called browser functions[1], that can be invoked from Javascript, and methods that can execute commands in the browser (see Figure 4.3). The Java code is structured as shown in Figure 4.2.

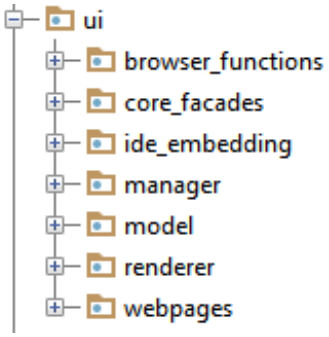

Figure 4.2: Package structure of the `ui` module

The `core_facades` package contains convenient interfaces, facade classes [GHJV95], for core functionality. For example, `ContactListFacade` encapsulates the four core classes, `ConnectionHandler`, `XMPPConnectionService`, `SubscriptionHandler`, `XMPPAccountStore` so that the browser function only has to deal with one dependency. When more functionality is included in the `core` module and the code of HTML prototype has become relatively stable, these methods could either move into the `core` classes, or into the `browser_functions` package if the facade turns out to be nothing more than a simple delegation.
The `model` package consists of GUI model classes, i. e. representations of the application state as it is rendered in the GUI. These classes can be converted into JSON strings and back into Java objects[2] for the exchange of data between Java and Javascript.
The classes in the `renderer` package are responsible for calling the adequate Javascript functions with the converted Java objects to display the application state in the browser. These rendering calls have to be made explicitly if the application state changes (see subsection 4.1.3).
The `manager` currently contains the `ContactListManager`, which listens for changes to the contact list. The `ide_embedding` contains the IDE-independent code to create

[1] In SWT they are instances of the `BrowserFunction` class.
[2] For this, the GSON library `https://code.google.com/p/google-gson/` (retrieved 20 March 2015) is used

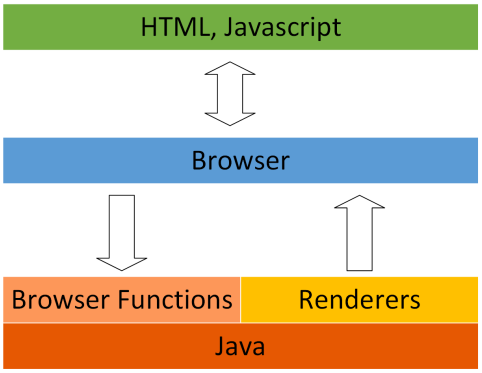

Figure 4.3: Bidirectional communication with the browser

the browser and dialogue. The `browser_functions` package contains the Java API which is callable from Javascript.
The `webpages` package contains abstractions for the different webpages, one for each dialogue page and one for the main view of Saros. Each class encapsulates the location of the corresponding HTML, the necessary browser functions as well as the renderers to be used.
Ideally the Java part of this UI layer is very small since it is just concerned with data conversion and the delegation of method calls which are implemented in the `core` module. Since the `core` module is independent of the availability of a user interface, it should contain as much functionality as possible.
In practice there are some problems that make the implementation of this small layer somewhat challenging:

- The management of two event dispatch threads.
- Returning the results of asynchronous calls back to Javascript.
- Reflecting the application state in the browser.
- The constant change of the browser instance.
- Validating user input.

4.1.1 Threading

There are two event dispatch threads to be managed, one for SWT and one for Swing – at least inside IntelliJ IDEA – and because the `ui` module must work in both IDEs, it has to deal with the fact that there might be multiple EDTs. As a result, the developer has to be conscious of which GUI toolkit the called methods and objects belong to. From my experience implementing the current code, this should not be a problem after a short period of orientation.
For all calls to IDE-specific GUI methods inside the `ui` module Saros' `UISynchronizer` class should be used, which abstracts the EDT of each IDE. Long running calls should be made asynchronously on a non-UI thread, so that the threading is similar to that of regular GUI development for Eclipse or IntelliJ plug-ins. The main change is that

all browser use must be done on the EDT.
In the adaptation of the browser component (see section 4.2) I implemented all browser methods in such a way that they can be called from any thread. They switch the thread internally if they are not already on the correct one. I had two motivations for this. Firstly, the calling code becomes shorter since potential UI thread calls are lengthy in Java. Secondly, the caller becomes independent of the used GUI toolkit so that the browser could be switched transparently. A negative effect is that this could make the developer unaware that a thread switch might happen and they still have to be careful about this.
In this context, it is also important to know that every Java code called from Javascript is executed in the SWT thread as illustrated in Listing 4.1.

```
new BrowserFunction() {
    @Override
    public Object function(Object[] arguments) {
     // this will be executed on the SWT thread
    }
}
```

Listing 4.1: UI thread context in browser functions

After parameter checks and possible conversions, in most cases a backend call is made. Here the programmer has to decide whether the call is short enough to be made in the UI thread or if the thread context should be switched. It is possible to enhance the `BrowserFunction` class to automatically execute the callback on a non-UI thread. The reason why I have not done this yet is that, for example, in the case of invalid input the browser functions will immediately report back to Javascript, so there would be an unnecessary back and forth thread switch. This might not be a noticeable problem but it is questionable to do this on an obligatory basis in the browser library. The easy solution would be to introduce two different `BrowserFunction` classes with only one switching to a non-UI thread, so that the programmer still has to make a conscious decision but the resulting code is shorter. This could be implemented when more people are developing the prototype and its actual requirements become clear.
In conclusion, the occurrence of two EDTs is manageable with a little thinking on the developer's side. As the HTML plug-in already uses a browser library that is under control of the Saros team (see section 4.2), common behaviour can be moved into the library so that the browser code in Saros becomes more comfortable.

4.1.2 Passing the result of asynchronous method calls back to Javascript

As explained in the previous section, Java method calls from Javascript might be executed asynchronously in a different thread. The question is how to inform Javascript about the result.

```
new BrowserFunction("function_name") {
    @Override
    public Object function(Object[] arguments) {
        new Thread(new Runnable() {
                @Override
                public void run() {
                    try {
                        longRunningCall();
                    } catch (RuntimeException e) {
                        LOG.error(e);
                        browser.run("alert('An error occurred.')
                            ;");
                    }
                }
            }).start();
        return null;
    }
}
```

Listing 4.2: Asynchronous execution in a browser function

In Listing 4.2 the content of the `function` method is invoked when the `function_name` method is called in Javascript. The execution happens on the SWT thread. Therefore, depending on the duration of the call, it might be desirable to make the call on a different thread in order to keep the application responsive. The problem is that by the time the browser function returns, the result of the `longRunningCall` method might not be available yet. Therefore, the success or failure cannot be signalled via the return value. The example always returns **null**.

```
new BrowserFunction() {
    @Override
    public Object function(Object[] arguments) {
        try {
            return shortRunningBackendCall();
        } catch (RuntimeException e) {
            LOG.error(e);
            browser.run("alert('An error occurred while...');");
        }
    }
};
```

Listing 4.3: Blocking of the UI thread

There are two different ways of dealing with this problem. First, one could decide to block the UI thread regardless. This is the easiest solution with fairly readable code and might not cause any noticeable delay in most cases. However, the developer has to be aware that the code in Listing 4.3 is executed on the UI thread and make the conscious decision that this is acceptable for the concrete call. Note that the return value is converted into a corresponding Javascript object but occurring exceptions are

not re-thrown in Javascript[3]. If an unhandled exception occurs, the value returned to Javascript is a string describing the error, which is useful for debugging at best but not for providing helpful feedback to the user. Thus, occurring exceptions are caught in the Java code and passed appropriately to Javascript. In Listing 4.3, an alert message is displayed.
Second, a more general approach is to define some sort of callback function in Javascript, which should be called as soon as the result of the operation is present. The obvious disadvantage – besides the verbose asynchronous call – is that the Java code calls a Javascript API beyond any compile time checks or IDE support. As a result, in course of the maturation of the HTML prototype an easy interface has to be agreed upon and well documented for such callbacks. In most practical cases, the solution will be as simple as calling `alert` or a similar method with an error message as parameter. I looked at a representative amount of the equivalent Saros/E code segments and all of them did this: validate the input, make the call (thereby mostly ignoring the return value), log occurring exceptions, and display a message to the user indicating whether something went wrong.

Example: Implementing a progress bar

I will now give the example of implementing a progress bar to illustrate the close connection between Java and Javascript beyond simple `alert` calls:
Javascript starts, for example, a file transfer by calling the appropriate Java method `__java_startFileTransfer('progressbarObject')` passing a reference to the Javascript object `progressbarObject`, which should be informed about updates on the progress. In reaction, Java will start a new task passing along the reference identifying the Javascript progress bar, e. g. the Javascript name as string. The executing Java code will eventually call

```
browser.run(progressObject.getJavascriptName()+".setProgress(val)")
```

This example illustrates that Javascript objects might have to be wrapped or converted into Java objects and passed between Java methods before converting them back and doing the actual callback. It further shows that Javascript objects like `progressbarObject` and method calls are just meaningless strings from a Java perspective.
This close connection between both languages is not a bad thing per se, nor is it unexpected because in order to make a rendering call you have to know the name of the responsible method regardless of the GUI toolkit. However, there are Java developers[4] who refrain from Javascript code and vice versa and that will not work well with the new GUI since Java and Javascript cannot be effectively separated. The missing look-up for Javascript method names in Java IDEs does not improve the situation. Generating Javascript code with identifiers known to Java might help here

[3] This is undocumented in the source code but is the currently implemented behaviour.
[4] To give a concrete example, in the code reviews in Saros a few people expressed a lack of experience for reviewing Javascript code.

but would add a considerable amount of complexity and thus should be postponed until the Java-to-Javascript interface has become more stable.
We have now had a close look at calls originating from the user interface (Javascript side). In the next section, we will examine the other direction: the display of application state in the user interface.

4.1.3 Displaying the application state

There are two different ways of getting the application state from the Saros core in order to display it in the browser; both are implemented in the current minimal prototype. There was no decision on how to implement which method; instead, the different approaches are a direct result of the `core` implementations and the way they provide data.

1. Query the core on each render call (it is possible to cache certain results). The `AccountRenderer` uses this approach. The state of the account list is managed by the `XMPPAccountStore` in the `core`. No events are fired when the state changes; instead, it can be queried via methods like `getAllAccounts()`.
2. Register a listener, reflect and store the state in the GUI classes. The `ContactListRenderer` uses this approach. For the contact list the querying approach does not work as changes do not originate at the front-end but at the other side of the network. Therefore, the `Roster` class supports listeners and informs them about updates.

Both approaches are viable. The first one has the decisive disadvantage that the programmer must recognise when the state has changed and explicitly query it. As Saros' current implementation allows the simultaneous use of the HTML and the traditional GUI, it is not possible to reflect an account change done in one view in a timely manner in the other one. So in this concrete case (and if the use of two simultaneous views is more than a temporary implementation) the introduction of an `AccountChangeEvent` – following the observer pattern [GHJV95] – has to be considered. Additionally, in the first scenario the browser functions for account creation have to know the rendering classes because they must trigger the re-rendering of the account list.
Because of the two different approaches, the implementation of the `ui` module is not uniform at the moment. While there is not enough core functionality available yet to make a general decision, I think that the GUI code might benefit from the consequent use of the observer pattern as it allows to completely decouple browser functions and rendering code and can support multiple simultaneous views.

4.1.4 Validation of Javascript input

In order to provide helpful information to the user, it is generally easier to validate the input beforehand instead of catching and translating thrown exceptions.
There are two locations where the validation can take place, in Javascript or in Java. For this decision, two competing goals have to be considered.

1. The validation code should be as close to the actual business logic as possible. Ideally, this implementation uses the same validation code for throwing exceptions.
2. The validation code should be as close to the user's input as possible in order to provide feedback more easily.

If the first item is neglected and the validation code is duplicated, it might diverge over time and fail to provide accurate feedback. Should this duplication stretch beyond the language border between Java and Javascript, it is even harder to keep both locations coherent as an IDE will not recognise the link.
For the second item, consider the small pseudocode example in Listing 4.4. The input consists of two fields and we want to highlight the one that does not pass the validation. This example shows that the validation can be done in Java without compromising the quality of the feedback.

```
<script>
if (!validateJID($("#jid").val())) {
  // highlight JID field
}
if (!validateAlias($("#alias").val())) {
  // hightlight alias field
}
</script>

<input id="jid" type="text" />
<input id="alias" type="text" />
```

Listing 4.4: Validation of input fields in Javascript

The `validateJID` and `validateAlias` can also be implemented in Java as long as they execute synchronously. In a real scenario the result of the validation would probably be an object transmitted as a JSON string and encapsulating the success flag and the accompanying reason. If the validation method is asynchronous for whatever reasons, Javascript has to provide a callback method (see subsection 4.1.2).
I exemplarily validated the `createAccount` method in the prototype. Through this, I noticed that the greatest difficulty is to re-write the `createAccount` method in the `core` to use the same validation because it should return a `ValidationResult` (value object [Eva03]) for the HTML code and at the same time still throw `RuntimeException`s for the existing implementation.

4.1.5 Changing browser instances

As explained in chapter 3, the closing of the Tool Window in IntelliJ IDEA disposes of the browser instance. This, however, does not change the application state of Saros and so the state of the GUI model has to persist as well. Therefore, it is undesirable to re-create the `Renderer` classes, which hold the current state. On the other hand, the rendering methods need to know the current browser instance and all browser functions need to be re-created after a change of the browser instance.

In the following I will outline the current implementation as well as some of the steps that got me there.
In Saros, nearly all of the core classes such as the `ConnectionHandler` or the `XMPPAccountStore` are created and managed by PicoContainer, a Dependency Injection container. As the UI classes require these components, the easiest solution is to let the UI classes be managed by the PicoContainer as well, so that their dependencies are automatically injected. As a result, the rendering classes are managed by PicoContainer and are effectively singletons, with their state persisting when the browser is re-created. Unfortunately, PicoContainer is unable to create and replace the constantly changing browser instances.
The next question is how the rendering classes acquire the current instance of the browser. The method making the rendering call does not know about the browser as I consequently separated the concerns of each class. Thus, only the `BrowserCreator` itself and the `Renderer` know about the browser. I introduced an abstraction for webpages, `BrowserPage` of Listing 4.5, and each instance knows its corresponding rendering classes and browser functions.

```
public interface BrowserPage {
  String getWebPage();

  List<JavascriptFunction> getJavascriptFunctions();

  List<Renderer> getRenderer();
}
```

Listing 4.5: Abstraction for webpages

This way the `BrowserCreator` can set the current browser instance in the `Renderer` and create the browser functions when it is loading a `BrowserPage`. The removal happens in the `DisposeListener` of the browser.

4.2 Extending the browser

As stated in subsection 2.2.3, Björn Kahlert's browser extension should be extracted into its own project and developed in close connection to Saros. This section describes my efforts to improve the browser component and make it usable as an external library.

4.2.1 The planned course of action

Initial state

The browser wrapper is bundled inside a set of widgets together with diverse utility functions for those widgets. It was developed to run under Eclipse on Mac OS and currently displays no browser on GNU/Linux and Windows. Because of many (nested) asynchronous calls and the multiple wrapping of `Runnables` and return values, the control flow is not easy to follow. I had no good overview as to which of

its features are working in an embedded environment on multiple operating systems yet or which are useful for Saros. Apart from this, the code looks well-written; it contains informative comments in the needed locations and features a few tests and demo applications, which can only be started in Eclipse.

My goals

My main objective is to extract the browser into its own repository[5] and make it an environment-independent library. For that, I will redefine and document the existing browser API with a focus on usability and currently required functions so that the details of the implementation can be changed later behind this interface[6]. Even though this API is meant to be relatively stable, it can be adapted when new requirements or different use cases emerge.
In addition, the functionality should be tested to get an overview of what is working already. Only currently required functionality must work initially but it is already certain that the necessary adjustments have to be made for Windows and GNU/Linux in order to get the browser running at all.

General Guidelines

As the browser wrapper contains a lot of knowledge that may be of use later, I will not delete (unused) functionality except when I fully understand its purpose and can think of no situation where it might be useful to us[7]. However, I will exclude unused or not-working methods from the public interface.
I will use small reversible commits targeting one issue. Especially, when removing any code, I use the `[REMOVAL]` tag in the commit message and do nothing else in the same commit.

Challenges

Since the browser must run under Mac OS, GNU/Linux, and Windows and some of its functionality is platform-specific, e. g. file handling, it should be tested (ideally continuously) on each operating system. As I have no constant access to Mac OS and the manual testing is time-consuming, I have to compromise. I will develop mainly under Windows and write tests and demos to run on the other systems at certain development stages.

4.2.2 Getting it to run on all systems

Removal of Java 8 features

Saros has the requirement to run under Java 6 and higher and therefore the browser library has to be compiled for Java 6. Thus Java 7 or 8 features must be removed

[5] On Github `https://github.com/ag-se/swt-browser-improved` (retrieved 20 March 2015)
[6] Due to time constraints I am only able to change certain parts of the code in this thesis.
[7] The code contained experimental parts and Eclipse-specific functionality.

again. For example, I converted all Java 8 lambda expressions, which Björn Kahlert had recently introduced, back into conventional anonymous classes.

Required fixes

For two reasons, there is no browser visible under Windows and GNU/Linux. First, the implementation makes the browser visible only after it has reported the page to be successfully loaded. As it tries to inject custom Javascript files and fails because of invalid URLs, loading fails, too. As soon as the Saros IntelliJ plug-in supported custom logging configurations, the browser produced error output and it was fairly easy to identify the problem. Before that, the browser remained silently invisible. The more obvious approach to debug it inside Eclipse remained fruitless as it would not start because of missing OSGI bundles which I could not fix even with Björn Kahlert's installation instructions[8].
Second, the way the browser is embedded into the Swing frame in IntelliJ it misses size or other layout information so that it is not rendered.
After I applied the fixes, I found a few places with similar file URL handling. So I went to make the classpath and file operations system-independent to prevent future problems.
For that, I replaced custom string manipulation like `new URI("file://"+ file.getAbsolutePath())` with standard Java API operations like `file.toURI();`.
I could eliminate all manual string manipulation in the other direction as well.

```
if (uri.startsWith("file://")) {
    return new File(uri.substring("file://".length()));
}
```

Listing 4.6: Manipulation of file URLs

The code in Listing 4.6 can be replaced by `new File(URI uri)`.
I had to re-write the affected method signatures to keep them consistent and eliminate unnecessary conversions back and forth. For that, I preferred meaningful classes like `URI` or `File` to simple `String` objects because they provide standard methods for the conversion.

Race condition causes JVM crash

When I was testing the Saros prototype under GNU/Linux, I noticed that the introduction of this browser library might cause the JVM to crash with a segmentation fault. This happens when a dialogue window is closed via the cancel button. However, it can take up to ten or more cancellations and re-openings of the windows until the error occurs.
It affects both Saros Eclipse and IntelliJ but only under GNU/Linux. As the segmentation fault happens inside C code, it is difficult to debug from Java. Fortunately,

[8] During my work I minimised the dependencies to external libraries and included the remaining ones in the deployable archive.

one of the SWT design principles is that there is one Java function for each native function[9], so that I could locate the calling Java code. The Java debugger, however, did not allow stepping up to this point but ended about two method calls away.
Since the error could not be reproduced reliably and debugging did not provide useful insights, I resorted to changing code and looking at the effects in the debugger. My best guess was to try and find differences in our code compared to the plain SWT browser, which does not exhibit this error. After I had unsuccessfully reworked the embedding procedure itself (the browser library uses an additional SWT `Composite` around the browser), I deactivated the Javascript event listeners.
While I am not one hundred percent sure what happens in detail, I confirmed that the `__focus_gained` and `__focus_lost` browser functions triggered the crash even though they are implemented as a no-operation if no listener is registered. Before clicking the "Cancel" button on the website, these Javascript event listeners call back into Java several times. Thus, when the browser is about to be disposed, there are some simultaneous calls to the browser. To dispatch the calls to it, the native function `_gtk_widget_get_parent` is called to find the browser. Apparently, this call is made without verifying that the corresponding location in memory has not been freed yet. I am unable to point to a concrete location where a fix could be applied but I think this is a bug in the GTK implementation of SWT. The currently applied workaround is to deactivate the focus browser functions as they are not being used yet.

Internet Explorer fixes

Even if the latest Internet Explorer (IE) version is installed, it may display the site in compatibility mode depending on the `<!DOCTYPE>` directive[10]. IE prior to Version 9 has issues with standard Javascript.
In the browser library, the affected IE versions fail to load the Javascript files for additional functionality (so-called event-catch functionality) such as listening for hover events and the management of the browser size[11]. I removed the errors so that the remaining parts of the injected Javascript files could be loaded, but I only back-ported functionality where it was easy to do so for those outdated IE versions. Because this functionality is not needed for Saros at the moment, I moved the event-catch functionality into a separate extension later on so that it can be deactivated easily (see section 4.2.6) if not needed.
In Saros we can code the HTML in such a way that new IE versions will not fall back, but users with old IE versions that are unsupported by Microsoft may still have compatibility issues. Whether they may experience concrete problems depends on Saros' actual Javascript implementation.

[9] `https://www.eclipse.org/articles/Article-SWT-Design-1/SWT-Design-1.html` (retrieved 31 March 2015)
[10] `https://msdn.microsoft.com/en-us/library/cc288325(v=vs.85).aspx` (retrieved 20 March 2015)
[11] Since I have the latest version of Internet Explorer installed, this is only noticeable on sites that trigger the compatibility mode.

Resource loading from JAR files

The browser's binary is distributed as a JAR file to be included in applications. As a result, the bundled HTML and Javascript resources are located inside an archive. These resources are loaded via file URLs, which does not work for files inside archives[12]. Therefore, I extended the resource loading methods to extract the content of the files into a temporary folder as an immediate solution. In a next step this could be improved to extract the files only once per application.

4.2.3 Forming the browser interface

I formed the public interface of the browser wrapper `IBrowser`. It contains mainly the following functionality in different variants:

- Loading a URL.
- Checking the loading state or waiting for a specific Javascript condition to be true.
- Injecting and executing Javascript and CSS code; the difference between injection and execution is that injected code persists and may be called from the website itself.
- Some utility methods for querying and setting parts of the HTML's DOM.
- Methods concerning the display and management of the wrapped browser, like `setSize()`, `runOnDisposal(Runnable)`, or `setFocus()`.

While consolidating the interface I was confronted with three major design decisions:

- How to handle the result of asynchronous calls.
- How to handle calls to blocking methods as they might happen on the UI thread.
- How to improve the exception handling, as the throwing and catching of the generic `java.util.Exception` class is ubiquitous.

Handling delayed execution

The main reason for the use of the browser component is its ability to delay method execution until a defined state is reached, in most cases this is the complete loading of the page. There are two different approaches to get the result of those delayed calls.

Blocking methods First, the calls can block until the execution has finished. This is the easiest solution as there is no extra code required to get the result or to be informed about exceptions[13]. However, there is one limitation: those blocking calls must not be made from the UI thread (EDT) as this reliably leads to a deadlock because the browser component needs the UI thread to complete the loading.

[12] Eclipse's `bundleresource:` URLs also have been considered.

[13] Even blocking methods might not give full success notification as Javascript might call other methods asynchronously.

Asynchronous methods The alternative is to make the methods execute asynchronously resulting in the question of how to get the result. The existing implementation returns instances of `Future`, which require the code in Listing 4.7 on the caller's side.

```
try {
    Future<Object> res = browser.asynchronousCall();
    res.get();
} catch (InterruptedException e) {
 // do something and call Thread.currentThread().interrupt()
} catch (ExecutionException e) {
 // all occurring exceptions are wrapped inside an
    ExecutionException
}
```

Listing 4.7: Using Java's Future class

This is tedious and requires the caller of the library to deal with the `InterruptionException` which is a common source for subtle errors[14]. Even explicitly dealing with the `ExecutionException` is an additional overhead because the browser only throws unchecked exceptions (see section 4.2.3).
Furthermore, this code only moves the blocking call. `Future#get()` is blocking and must again not be called on the UI thread. Björn Kahlert implemented an own `Future` class that throws an exception when its `get` method is called on the UI thread. Otherwise the code would silently deadlock. While this is definitely an improvement, the real problem remains. Calling these methods is uncomfortable; imagine the extra code required to make the `res.get()` call in Listing 4.7 in a new thread.

Callback functions One way around this is the use of callback methods. Generally speaking, the blocking call is done in another thread and calls a method after it has finished.
I introduced the parametrised interface of Listing 4.8 where `T` is the type of the return value of the callback function and `V` is the return value of the asynchronous method.

```
public interface CallbackFunction<V, T> {
    T apply(V input, Exception e);
}
```

Listing 4.8: Parametrised callback function interface

If the method throws an exception, it is passed to the `apply` function via the second parameter.
As one of the major applications for the callback will probably be error logging, I already predefined the static error logging callback function shown in Listing 4.9.

[14] The browser wrapper implementation in most parts violated the contract and consumed the `InterruptionExceptions` (see [GBB+06] for more information).

```
public static final CallbackFunction ERROR_LOGGING_CALLBACK =
    new CallbackFunction() {
     @Override
     public Void apply(Object input, Exception e) {
         if (e != null) {
             LOG.error("Error in async call: ", e);
         }
         return null;
     }
};
```

Listing 4.9: Callback function to log errors

Why mostly error logging? One of the problems with asynchronous method execution is that exceptions can easily get swallowed because, if the caller is not explicitly interested in the return value, they might neglect to call `Future#get()` just to check for exceptions[15]. Now, by just adding `ERROR_LOGGING_CALLBACK` as second parameter, at least errors get logged. Since nearly all of the exceptions reported back to the user are not meant to be handled (they mostly indicate programming errors), logging (and informing the user) is the reasonable thing to do in most cases. Other callback methods can be added for different behaviour[16].

Resulting interface Depending on the concrete situation, either of the three variants blocking call, asynchronous call with `Future` and with callback can be the most suitable one. Therefore, I added all three variants for all the different method types to the interface.
Furthermore, to reduce the number of asynchronous calls to a minimum, the user may now even call potentially blocking methods on the UI thread if they make sure that the condition is already met. I changed all blocking methods to check whether they can be executed immediately because they do not have to wait for the condition. However, if they would block the UI thread, they throw an `IllegalStateException` as documented in their contract. In practice this may avert the previously described problems altogether because, for example, once the loading is completed, no future calls will block and most Javascript calls will be made long past the loading process.

Busy waiting on the UI thread In theory it is possible to busy wait on the UI thread. For example, Björn Kahlert implemented the method of Listing 4.10, which is unused. I adapted it slightly to make the mechanism more explicit.

[15] I explained how uncomfortable this call is.
[16] If an application shutdown should be attempted on certain errors, the developer can also do so inside a callback method.

```
public static Object busyWait(final Callable<Boolean> whileTrue)
    {
     new Thread(new Runnable() {
         @Override
         public void run() {
             try {
                 while (whileTrue.call()) {
                     Thread.sleep(20);
                 }
             } catch (Exception e) {
                 LOGGER.error(e);
             }
             conditionIsMet.set(true);
             ExecUtils.asyncExec(new Runnable() {
                 @Override
                 public void run() {
                     // Should there be no more events on the
                        queue, the loop
                     // below will never stop. Let's make it work
                          again with
                     // this Runnable.
                 }
             });
         }
     }).start();
     Display display = Display.getCurrent();
     while (!display.isDisposed()) {
         if (conditionIsMet.get()) {
             return result;
         }
         if (!display.readAndDispatch()) {
             display.sleep();
         }
     }
}
```

Listing 4.10: Busy waiting on the UI thread

This implements the event dispatch functionality again while it is checking the completion in defined time intervals. Replacing the event dispatch loop at runtime is a little ugly but has no real negative impacts besides code duplication. Whether the overhead of the constant busy waiting calls has noticeable impacts on performance would have to be tested. Doug Lea ([Lea00] chapter 3.2.6) discourages own busy waiting implementations. Again having this workaround in the framework does not seem to reasonable just for relieving the programmer of the need to think about UI threads. I listed this alternative here because it may be of interest some day.

Side note about the callback in the API The *guava*[17] library, which contains all kinds of useful functionality and which I originally added for the uniform checking of parameter passed to the public browser API, defines a `ListenableFuture` interface. Classes implementing this interface allow to listen for the completion of events and to register callback functions. Basically the return of `ListenableFuture`s in the browser API would make the additional methods with callback function parameters obsolete and thus the browser API more concise. However, this interface has the `@Beta` annotation and the developers discourage the use of such classes in public API as they might change completely. Looking at the current development and the fact that the interface has been around for years, I find this unlikely. Nevertheless, the current solution lacks no features, it simply has more methods.
Summarising the point of this side note, if the `@Beta` annotation is removed in the future, the re-use of functionality and the concise interface have to be weighed against the exposure and consequently heavy dependence on a third-party library.

Exception handling and throwing

The existing implementation makes heavy use of the generic `Exception` class which is inconvenient for the user. The cause for this is probably that the `call` method in Java's `Callable` interface throws a generic exception[18].
Therefore, I had to remove at least those generic exceptions from the public API and replace them by specific exception types. The main decision was which types of exceptions to throw.

Checked vs. unchecked exceptions The bottom line guideline for the use of checked vs. unchecked exceptions is the following: "If a client can reasonably be expected to recover from an exception, make it a checked exception. If a client cannot do anything to recover from the exception, make it an unchecked exception"[19]. As exceptions inside browser are most likely the result of a programming error either in the browser library or in the Javascript code supplied by the user, the choice was to use unchecked exceptions.
The browser loads content from streams and files so the checked `IOException` can be thrown. `IOException`s occurring while loading browser-internal files are wrapped inside unchecked exceptions because there is nothing the caller can do about them (besides filing a bug report)[20]. For `IOException`s that are the result of a wrong external file URL, the decision is the same and deviates from Java's decision to make

[17] `https://code.google.com/p/guava-libraries/` (retrieved 20 March 2015)
[18] I introduced a derived interface which overrides the `call` method to remove the **throws** clause in order to get rid of the constant need for dealing with exceptions inside the library code that are never thrown anyway.
[19] `http://docs.oracle.com/javase/tutorial/essential/exceptions/runtime.html` (retrieved 20 March 2015)
[20] They might be the result of a wrong usage of the browser library but even then they cannot be handled at runtime.

the `IOException` a checked exception[21]. In the concrete case of loading a URL, the success or failure is indicated via the return value and not via an exception.

Concrete implementation The browser API encapsulates the multitude of internal exceptions by returning the unchecked `JavascriptExecutionException` which is at the appropriate level of abstraction ([Blo08] item 61: " Throw exceptions appropriate to the abstraction").
All plainly wrong usage of the browser API, such as **null** values are immediately answered with the corresponding unchecked exceptions ([Blo08] item 60 " Favor the use of standard exceptions").
All possibly occurring exceptions are documented in the public browser interface and provide their full stack traces. Consequently, the client of the browser can fulfil its responsibility to hide them from or show exceptions to the end-user.

4.2.4 Internal Concurrency fixes

Thread Executor

The browser uses the generic `ExecUtils` class containing static methods for the synchronous and asynchronous execution on both the UI thread and non-UI thread. Furthermore, this class features variants of these base methods to support timed execution and thread labelling.
I initially planned to view the class as a complex black box and keep the redundancy between the individual methods. However, it contains a static instance of `ThreadPoolExecutor` that is started during class loading but is never explicitly shutdown. Registering a JVM shutdown hook does not work inside a plug-in because the browser might close long before the whole IDE exits. The problem became apparent while running the demo applications (see subsection 4.2.5) as the JVM did not stop after closing the browser.
Another issue is that the UI thread execution is global and cannot be replaced at runtime when a different browser should be wrapped which is a likely scenario in the future.
Fortunately, both problems can be addressed simultaneously. First, I separated UI and non-UI execution methods and made the individual variations of the same method type call each other to remove redundancy. To eliminate this redundancy it helped to hide the thread labelling code behind new abstractions, specialised implementations of the `Callable` class.
In the end, I could make `ExecUtils` an instance of `ThreadPoolExecutor` because the formerly method static methods basically just decorated calls to the `ThreadPoolExecutor`. Therefore I extended the `ScheduledThreadPoolExecutor` class, which conveniently provides extension points to wrap input and output of the `execute` methods, and had to include just two new methods signatures for the

[21] This does not mean that it was wrong to make `IOException` checked, but in our concrete use case an unchecked exception is more appropriate.

thread labels. The use of the `ScheduledThreadPoolExecutor` got rid of the re-implementation, calling `sleep`, of its delayed execution functionality.
With the elimination of the static state both non-UI thread executor and UI thread executor can become properties of the browser. As a result, the browser is responsible for creating and terminating the `ThreadPoolExecutor`, which is done in the browser's dispose listener.
In addition, the browser can decide which instance of the `UIThreadExecutor` it creates. The SWT browser implementation creates an `SWTUIThreadExecutor` whereas the JavaFX browser instantiates a `JavaFXUIThreadExecutor`.
Even though the use of the `ScheduledThreadPoolExecutor` seems logical, it has one decisive disadvantage: it is a fixed size thread pool. Since the browser executes tasks that depend on each other, it might deadlock if the pool size is too small[22]. Basically there is one task waiting for the completion of the page load while some tasks are loading the page. Apart from the overhead of having idle threads in a fixed-size thread pool the number of executing tasks is theoretically unbound[23] as the user may supply custom tasks that are executed before the loading is complete.
I did two things to remove this issue. First, I removed the timeout task for the page loading from the `ScheduledThreadPoolExecutor` and used Java's `Timer` class instead, which offers the same functionality outside of a thread pool. As this was the only task using the schedule function, I could switch back to a regular `ThreadPoolExecutor`, which Java also provides with variable size. Second, the waiting task is now executed outside the thread pool. As it is relatively long-running and mostly executed once, this is sensible even from a performance perspective.

Additional concurrency fixes

During the implementation I introduced and extended the synchronisation of shared mutual state, e. g. for the management of the browser state and the management of listeners. I documented places that I deemed thread-safe but I am also pretty sure that there are still locations with unguarded state (due to time constraints).
In addition, I began implementing a cancellation policy. This mainly means that I fixed the swallowing of `InterruptedException`s so that all methods follow the basic principles for responsive applications.

4.2.5 Tests and Demos

I made the existing demo classes startable outside of Eclipse. These demos contain a few buttons to execute functionality such as the injection of Javascript or provoking Javascript syntax errors. They can be used to easily test fundamental features of the browser on multiple platforms. They are a good replacement for considerably more expensive (regarding development costs) automatic GUI tests. As there is no

[22] [LMS+12] "TPS01-J: Do not execute interdependent tasks in a bounded thread pool"
[23] Before this design flaw has been addressed, five should have been sufficient in practice.

infrastructure to support GUI tests on multiple operating systems yet, they would have to be installed and started manually on each system anyway.
I also wrote automatic unit tests for the loading of websites and for the Javascript execution methods. Thereby, I noticed that the `runImmediately` methods reliably failed when called directly after loading a URL because those methods do not wait for any condition, they just fire the Javascript code. As they provide no guarantees about the result of their execution I removed them from the public interface. The newly introduced `syncRun` methods (see subsection 4.2.3) can be used as replacement. However, these methods may block if the browser is not initialised.

4.2.6 Restructuring and decoupling functionality

Browser and BrowserScriptRunner class

The browser contains the state machine shown in Figure 4.4. It does not support the sequential loading of different sites in the browser, which Saros might need to display multiple-page wizards.

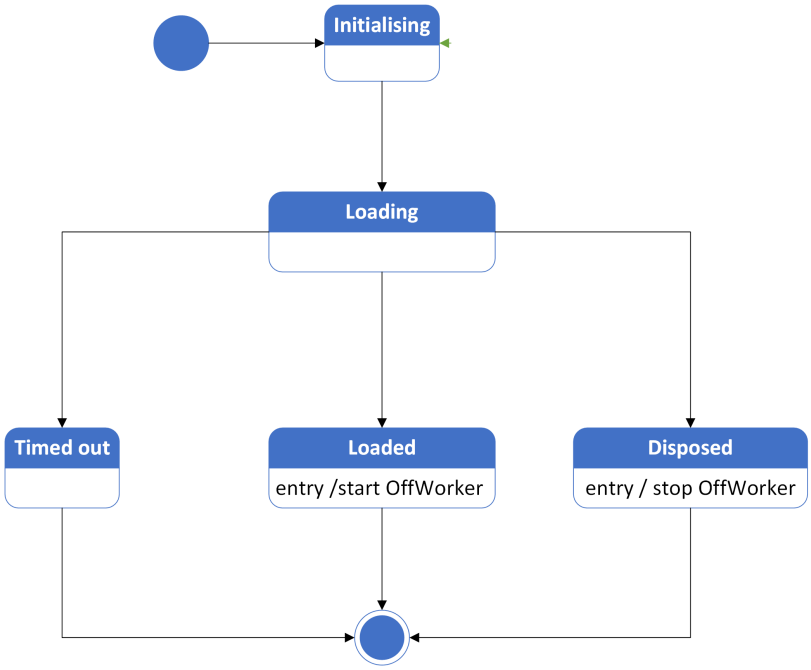

Figure 4.4: The browser states

Although the addition of an extra state transition is not too complicated, it has subtle consequences for the management of the queue for delayed tasks, which is the responsibility of the so called `OffWorker` class (see the next section).
While reading and getting to know the browser code, I moved responsibilities between classes to provide a clearer structure and make classes smaller.

For example, I extracted the browser state flag and the related (scattered) code from the `BrowserScriptRunner` into an own class, `BrowserStateManager`, that is now a property of the browser.
The existing separation in `Browser` and `BrowserScriptRunner` seems artificial because on the one hand `Browser` wraps `BrowserScriptRunner` and delegates some methods directly to it but on the other hand the `BrowserScriptRunner` also holds a reference of the `Browser`. Furthermore, it looked a little inconsistent as to which methods were implemented in which class. Thus I removed the separation in a first step to introduce a different one in the following step. This time I separated according to the following idea: there is an `InternalBrowserWrapper` which just adds the fundamental functionality to the SWT browser, being:

- Support for reliable detection of page loading and delayed method execution.
- Decoration of executed Javascript string with meaningful exception handling.

The `Browser` class in turn wraps this `InternalBrowserWrapper` and provides a rich interface – implementing the external browser API – consisting of methods with different parameter types, which use the few base methods. The connection between `Browser` and `InternalBrowserWrapper` is now unidirectional.

Simplified OffWorker

The `OffWorker` possesses different states as well. They are illustrated in Figure 4.5.

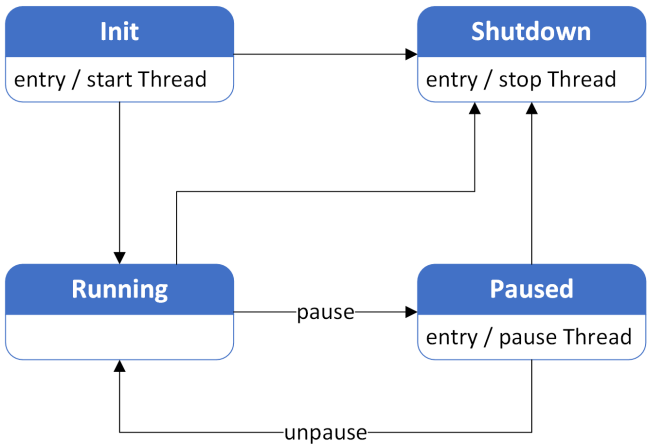

Figure 4.5: The states of the OffWorker class

Before my extension of the browser's state machine to support multiple loading cycles, the `OffWorker` was just a one-way train. It was initialised, started, and shut down. The pause functionality was not used for the browser and so I removed it to keep the class simple. This lifecycle can also be emulated by starting an internal thread without differentiating any explicit state, which simplifies the logic. When an `OffWorker` instance is no longer needed, the thread is simply terminated. This also got rid

of problems like “Cannot switch state from RUNNING to RUNNING”, which were probably caused by insufficient synchronisation.
After my extension, the `OffWorker` could either manage an explicit state again or just be recreated for each loading process, which is the solution I preferred as this keeps the code concise. There is no need to manage a state explicitly as each thread (or object) can be started and disposed of.
However, there is still one minor problem concerning the clearing of the task queue when loading a different webpage. Before the new document starts loading, the currently executed task has to be either cancelled or has to finish (not yet started tasks can easily be deleted) in order to prevent an old Javascript command from being sent to the new HTML document. A correct implementation is a little tricky (regardless of how the `OffWorker` is actually implemented) due to a possibly asynchronous task execution and because Java only supports cooperative cancellation. As this problem can only occur if the website is switched before it has finished loading, I have postponed the realisation.

Extracted Event-catch functionality

The browser features the registration of listeners for Javascript events, like hovering of anchor tags or elements gaining and losing focus. As already explained, this functionality is not yet needed by Saros and may have compatibility issues with old versions of Internet Explorer (see section 4.2.2).
For this reason, I encapsulated this functionality into a separate browser extension so that it can be easily deactivated. The browser already contained an extension mechanism; for example, there is a `JQueryExtension` that makes the JQuery library available. I bundled the separate browser extensions into one `enum` class, with the properties shown in Listing 4.11. An extension basically describes a list of files that have to be loaded and a Javascript expression that is used to check if the extension has already been loaded.

```
public enum BrowserExtension {
    [...]
    private final String name;
    private final String verificationScript;
    private final List<File> jsExtensions;
    private final List<File> cssExtensions;
    private final List<BrowserExtension> dependencies;
}
```

Listing 4.11: Browser extension

With the help of the defined extensions, specialised versions of the browser interface can be created, like the new one for the event catch functionality in Listing 4.12.

```
public interface IEventCatchBrowser extends IBrowser {
    void addAnchorListener(IAnchorListener anchorListener);
    void addMouseListener(IMouseListener mouseListener);
    void addFocusListener(IFocusListener focusListener);
    [...]
}
```

Listing 4.12: Interface of the event catch browser

4.2.7 Preparing the replacement of the underlying browser

JavaFX

Since it is currently uncertain whether the SWT browser will run inside IntelliJ IDEA on Mac OS with Java 8, its use might have to be re-evaluated in the near future.

As I already explained in subsection 2.2.5, the JavaFX browser seems to be a viable alternative because it is fully supported from Java 8 onwards on all platforms. Therefore, while doing the necessary changes, I already encapsulated all SWT related functionality in separate classes, which form a parallel class hierarchy, and experimentally included the JavaFX browser.

The long term goal is to provide multiple browser implementations behind the same public browser API so that the change of the browser is in theory as easy as calling a different factory method.

In practice there may be cross-browser issues and the JavaFX browser provides a more sophisticated Java-to-Javascript connection so that it might be cumbersome to wrap it behind the SWT browser-oriented API. However, for a first step it should be practical and the code can evolve once it works.

One example for the more sophisticated communication are the browser functions. While SWT only allows defining Javascript functions in the global namespace, JavaFX requires the definition of Java classes that are mapped to Javascript objects with a corresponding function defined for each Java method. Thus, JavaFX cannot define Javascript functions in the global namespace. I wrote a work-around to enable the definition of top-level Javascript functions in JavaFX, so that the client's Javascript code does not have to change. The cleaner solution, however, would be to convert the client's Javascript calls as the use of the global namespace is discouraged in Javascript. The conversion can be done automatically with a script or a simple search and replace.

All in all, I introduced three interfaces that have to be implemented when a new browser is introduced:

- `JavascriptFunction` to abstract from SWT's `BrowserFunction` class.
- `UIThreadExecutor` (see subsection 4.2.6) for the different UI threads.
- `IWrappedBrowser` for the actual browser implementation.

JxBrowser

As soon as I had formed the `IWrappedBrowser` interface, I used it to let the browser library use the JxBrowser internally. As the APIs of JxBrowser and SWT browser are very similar, this was straightforward. The only thing to add were conversion methods because the JxBrowser returns instances of `JSObject` from Javascript commands and provides no conversion to basic classes like `Double` or `String`.

4.3 Design of the GUI test framework

This section is concerned with IDE-independent GUI tests. First, I describe the existing STF implementation and then explain the considerations for an HTML adaptation.

4.3.1 The existing test framework for Eclipse

Saros has a GUI test framework named STF (Saros Test Framework), which is for Eclipse only. I will now shortly summarise its current mechanics and functionality. More detailed information can be found in the three theses [Szü10], [Che11], and [Ros11] (see section 1.6).
STF is basically a wrapper for the SWTBot and provides its methods via a Java RMI interface. SWTBot is a GUI automation library for the SWT toolkit and also features methods specifically for the Eclipse IDE. STF is divided into three so called Bots: `ControlBot`, `RemoteWorkbenchBot`, `SuperBot`. Each one is suitable for a different situation:

- `ControlBot` does not use the GUI, but rather accesses the business classes directly to quickly provide a defined state for testing. It can create accounts, for example.
- `RemoteWorkbenchBot` and its subset `RemoteBot` simply wrap their SWTBot counterparts and provide access to each individual GUI widget, e. g. a button.
- `SuperBot` provides a higher-level abstraction for `RemoteBot` and is more use-case oriented; for example, it allows adding a contact with just one method call, which translates into multiple `RemoteBot` calls.

If configured accordingly, Saros runs an RMI server at start-up and provides access to those Bots. The GUI tests connect to the RMI server, acquire instances of the exported objects and can call the desired methods to emulate user interaction.

4.3.2 Goals

My goal is to provide a similar framework for the new HTML GUI for both IntelliJ and Eclipse. It is to be decided whether the same interface as in STF will be used.

4.3.3 GUI automation for IntelliJ IDEA

Currently Saros/I offers no GUI test functionality whatsoever. The IntelliJ plug-in development API only offers headless integration tests[24].
Since IntelliJ uses Swing for its GUI elements, an equivalent to the SWTBot for Swing would be useful.
Internet research turned up two candidates UISpec4J[25] and Fest[26]. QFS[27] is a commercial alternative that will, for that reason, not be mentioned further. Neither of the first two candidates are actively developed but they seem mature enough to be an option. The major shortcoming of both is that they are aimed at regular Swing applications and have no equivalent to the `WorkbenchBot`, which knows the Eclipse workbench elements. As a result, it is quite complicated to automate even simple clicks in the IntelliJ GUI, for example opening a Tool Window.
However, at the current stage of development, we just have to be able to open the Saros Tool Window such that the browser becomes visible. All further GUI test commands can take place inside the browser. For the complete implementation of the GUI test framework other IDE-specific elements like the editor view, the project tree, and the settings menu might become relevant. Then we have to look at the concrete use-cases to decide whether the application of one of those GUI automation libraries is possible or whether it is more practical to obtain the desired information below the UI layer.

4.3.4 Opening the browser view in IntelliJ

As the only currently needed functionality is the opening of the Tool Window, I will concentrate my efforts on implementing it in both frameworks. This is exemplary for the implementation of further IntelliJ-specific functionality. Since it involves clicking nested pop-up menus, it is probably one of the harder automation tasks.
To open the Tool Window the easiest (not the only) way is to open the “View” menu, click “Tool Windows” and select “Saros browser”. I did not manage to do this using UISpec4J within a timeframe of two days. I can only speculate about the reasons: UISpec4J normally requires to be started before the Swing application to be able to intercept frames or pop-up windows. This is not possible in our scenario as it is launched inside a plug-in. The documentation is sparse and all further inquiries would require deeper debugging of the source code.
Using fest-swing I succeeded with limitations. If started in a multi-display set-up, the controlling and the controlled IntelliJ instances have to be on the same screen and the movement of the mouse occasionally disturbs the robot.
I further implemented a method to check whether the Tool Window was open both by looking at the Swing component tree and by accessing the IntelliJ API to obtain

[24] This functionality cannot be used by Saros/I because of a version clash of two PicoContainer versions.
[25] `http://www.uispec4j.org/` (retrieved 20 March 2015)
[26] `https://code.google.com/p/fest/` (retrieved 20 March 2015)
[27] `http://www.qfs.de/en/qftest/index.html` (retrieved 20 March 2015)

the information below the UI layer.

4.3.5 Opening the browser view in Eclipse

On the Eclipse side, the existing methods of the `WorkbenchBot` are simply re-used in order to open the browser view.
Once the view is open, we can focus on the actual testing of the Saros HTML UI; this is described in the following section.

4.3.6 GUI automation for HTML

First of all, standard web automation frameworks like Selenium webdriver[28] or Watir[29] cannot be used as they need to render the webpage themselves. Even if Saros could run standalone and listen on a port, the GUI would be tested outside the IDE which leaves out the entire browser integration and would not be a real GUI test. Currently this integration is a hot spot for stability issues.
Fortunately, automation inside a website is fairly easy as Javascript can be executed to emulate user input. To find individual elements, it is useful to give `id` and `class` attributes to the HTML tags. Then, the actual call is as simple as executing `$('#id').click()` or `return $('#id').text()` in JQuery syntax.
The integration in Saros and the design of the RMI interface pose more questions and are discussed in the next section.

4.3.7 Design of the RMI interface for HTML

The first option to evaluate is the re-use the current STF interfaces. I therefore gave them a close inspection.

STF's current interfaces

`RemoteWorkbenchBot` and `RemoteBot` are one-to-one wrappers of the SWTBot with some widgets excluded, such as the browser. Consequently, `RemoteBot` provides about eleven to twenty look-up methods for each of the sixteen supported SWT widgets. This look-up can happen according to ID, label, index, group or a combination of these, for example. Listing 4.13 shows just a very small section of this interface to give an impression. `RemoteWorkbenchBot` extends the list of widgets by offering access to Eclipse workbench elements like perspectives, views, editor. Both the bots use the page object pattern[30] and provide an additional interface for each of the widgets.

```
public interface IRemoteBot {
[...]
    IRemoteBotButton buttonWithLabel(String label);
    IRemoteBotButton button(String mnemonicText, int index);
```

[28] `http://www.seleniumhq.org/` (retrieved 20 March 2015)
[29] `http://watir.com/` (retrieved 20 March 2015)
[30] `https://code.google.com/p/selenium/wiki/PageObjects` (retrieved 20 March 2015)

```
    IRemoteBotButton buttonWithTooltip(String tooltip);
    IRemoteBotButton buttonWithId(String value);
    IRemoteBotButton buttonWithLabelInGroup(String label, String
        inGroup);
[...]
}
```

Listing 4.13: Extract from RemoteBot's interface

Adaptation for HTML

How does this fit to HTML? At least for some widgets, e. g. `Button`, `StyledText`, and `List`, corresponding HTML tags can be found. Even though for the mapping of HTML tags to Java objects a framework like selenium would have been of help, it is quite possible to manually convert Java method calls to Javascript strings representing the according operations in the browser.
So only a subset of widgets can be re-used in the HTML context. What about the accessor methods? Except for the by-group selector, all of them could be implemented in HTML, but only selecting by ID and visible text are really needed. The index is an unreliable way to find an element, especially in the face of the alternatives. The variety of CSS selectors is big but, for example, selecting according to attribute values or parent elements requires detailed knowledge about the implementation.
As a result, the concrete interfaces should be changed to reflect the different HTML widgets and the number of selectors should be reduced to selection by ID and text in the first draft.

Mapping between Java and HTML There are two ways to acquire the HTML state. The first parses parts of the HTML DOM and stores the state in Java objects. This might be more efficient than the second approach below as more information is transferred with one call. Instead of the three calls `$(object).getAttrX()`, `$(object).getAttrY()`, and `$(object).getAttrZ()`, one `parse($(object))` call could suffice to transfer the information to a Java object. However, even with libraries like jsoup[31] it is cumbersome because HTML attributes may be added and changed dynamically via Javascript. The main problem is that the state might be stale at the time the information is queried from the Java object.
Therefore, I will resort to the second approach and only build the Javascript strings and execute them directly when the call is made.

Representing structured widgets That was the easy part: simple, self-containing HTML elements like a button can be identified via ID. But what about lists? There is a list widget in SWT and there are lists in HTML. However, the concrete structures of what looks like the same list in the browser might be considerably different. To begin with, formatting in HTML makes heavy use of introducing extra tags just for

[31] `http://jsoup.org/` (retrieved 20 March 2015)

positioning or styling. Listing 4.14 shows an example of an HTML list as it can be found in a textbook, while Listing 4.15 shows the same list structure with additional HTML tags.

```
<ul>
 <li>item</li>
 <li>item two</li>
<ul>
```

Listing 4.14: A standard HTML list

```
<ul>
 <li><span>item</span></li>
 <li>item <span>two</span></li>
<ul>
```

Listing 4.15: A alternative representation of a list in HTML

An HTML list may not be built with `<ul>` at all, like the one in Listing 4.16.

```
<div>
 <span>item</span>
 <span>item <span>two</span></span>
</div>
```

Listing 4.16: A list structure in HTML with explicit list elements

Especially when using web frameworks we might not even have control over how the list is implemented. So what is the way around? I would propose the use of the same CSS class attribute for all elements of a list. Listing 4.17 shows an example for this.

```
<div>
 <span class="list1">item</span>
 <span class="list1">item two</span>
</div>
```

Listing 4.17: Identifying list element via the class attribute

This at least somewhat abstracts from the concrete implementation. However, the problem of individual structured list elements still persists. Now, changes to the general list structure with a multitude of completely different approaches have no effect on the Java-to-HTML mapping, only changes on the list element level still do. Listing 4.18 shows an example for a possible contact list. The `class` attribute identifies the individual contacts representing one list element each.

```
Contact list:
<div>
 <span class="list1">Alice <emph>online</emph></span>
 <span class="list1">Bob</span>
</div>
```

Listing 4.18: A more complex list example

As the styling can be done via CSS independent of the structure, this is somewhat immune to changes. However, at least when a new element like the subscription status is added or when parentheses are added around the status, the code of the test framework will still have to change.
Of course, now a list is not a general widget with a uniform structure anymore, instead there is a contact list widget with a certain structure and probably an account list widget with a different structure. It makes complete sense for a Saros-specific test framework, though, to know Saros-specific presentation structures.
Looking at the current Saros/E implementation, there are only a few structures of interest:

- The contact list.
- The account list (in the drop-down menu).
- The session tree (may be split into sub-structures).
- The chat, which contains a few elements and a list of structured lines.

Therefore, having general-purpose interfaces would add unnecessary complexity and is less direct to use. For example, now we can say `getContactList()` instead of `getList("contactListId")`, saving one layer of abstraction and more importantly getting exactly the specific elements needed, like `name` or `status` instead of a `StyledText` widget.
For self-contained widgets such as buttons or texts the current approach is fine as there are many more of them and they all have the same structure.

Summary In conclusion, for the first implementation I will use a new interface instead of the `RemoteBot`. For simple widgets it is inspired by the existing one but much leaner containing only the necessary selector methods, for complex widgets like lists I will define content-sensitive classes to reflect the Saros GUI.
The `ControlBot` interface can be used as-is because it is independent of the GUI.
The `SuperBot` interface aggregates multiple `RemoteBot` functions into use-case oriented method calls, which makes sense in HTML, too. However, the concrete interface is very selective, containing functions as they were needed at the time by the existing tests. So the current form is not very suitable for the limited HTML GUI, but can be expanded organically.

4.4 Accompanying refactorings

Before PicoContainer could be used in the `ui` module (see subsection 4.1.5), I had to move the `SarosPluginContext` class into the `core` and introduce a new factory class that creates the PicoContainer components for the UI classes. `SarosPluginContext` is necessary to inject dependencies into instances that initially cannot be created by the PicoContainer. While I was moving the `SarosPluginContext` class into the `core` module, I decided to remove further redundancy and moved the `SarosContext` class and the `SarosCoreContextFactory` class there as well. As the existing constructor of `SarosContext` expected nested `ISarosContextFactory` classes, which does not

reflect the non-hierarchical structure, I changed it to accept a list of factories instead. This also eliminated the need for the same logic in every factory to load nested contexts.
To avoid introducing new redundancy during the implementation of the HTML GUI, I extracted the account look-up method and two connection establishment methods out of the IntelliJ GUI classes (they clearly do not belong there). I then moved the account look-up method into the core class `XmppAccountStore` together with new test methods.
The connection methods were temporarily located in the `ui` module until Arndt Lasarzik moved the `ConnectionHandler` into the core so that those temporary methods could be removed.

4.5 Integration of the HTML GUI into the build process

The new `ui` module with the HTML GUI had to be integrated into Saros' build process in three places: in both IDEs and in the Jenkins build.

4.5.1 Building inside Eclipse

For the HTML GUI to be compiled in Eclipse, not only does the module itself require the appropriate `.project`, `.classpath`, and `MANIFEST.MF` configuration files, but the dependent module `eclipse` (compare section 4.1) also needs to be adapted to find and use the `ui` module. These modules need the correct OSGI bundle configurations, e. g. exporting the appropriate packages, to work together at runtime.

4.5.2 Building inside IntelliJ

I created the IntelliJ module configuration for the `ui` module, which mainly consists of the build paths and the required libraries, so that it gets built in IntelliJ as part of the Saros project.
A special challenge was to require as few manual steps as possible to get the Saros project running in IntelliJ after an initial checkout from the git repository. Due to license conflicts, Saros does not provide the required SWT library, so that each developer has to download it themselves (see section 2.2.2). Now, the library is already configured in the IntelliJ project, so that they just have to copy the downloaded JAR file into the `lib` folder. This way, each developer can use the same IntelliJ configuration files and those can be shared in git without having local changes.

4.5.3 Configuring the Jenkins build

ANT build files The continuous integration server Jenkins builds Saros using the Ant build tool. Therefore, I had to adapt the `build.xml` files for the `core`, `eclipse`, and `ui` modules. The IntelliJ one was especially cumbersome because it had been automatically generated resulting in the overly extensive declaration of Ant properties

and then adapted by hand. The one for the `ui` module had to be done twice as the Eclipse implementation was completed about two months after the IntelliJ one and it turned out that the resolution of Eclipse dependencies is not practical without using a specialised Ant-task (`ant4eclipse`).

Jenkins configuration The special challenge here was that Saros has two differently structured Jenkins jobs for Gerrit and for the master branch. As the job for the master branch uses a downstream job for the IntelliJ part, the artefacts from the Eclipse job, which builds the `core` and `ui` module, have to be copied. However, now the IntelliJ job would need to extract these archives inside the Ant code in order to load the required classes or – the current pragmatic solution – build `ui` and `core` again and use the copied artefacts just for the included third-party libraries.

5 Being part of the Saros team

This chapter describes the work I did apart from the main topic of my thesis as part of the Saros team. It further contains retrospective thoughts on my major design decisions and suggestions for an improvement of Saros' processes.

5.1 My contribution

5.1.1 Cooperation with parallel theses

Since the Master's theses of Matthias Bohnstedt and Bastian Sieker both build on my results, I explained my implementation and the ideas behind it to them. We often exchanged our thoughts and gave valuable input to each other in and outside of the review system. For example, I could profit from Bastian Sieker's experience with Javascript technologies.

GitHub set-up for the evaluation phase

During the first phase of the evaluation, in which Matthias and I tried to embed many different browsers into both IDEs, I set up a GitHub project for this cooperation. It required a somewhat complicated configuration, because we wanted to integrate it with the existing Saros plug-in early and the IntelliJ implementation had not been fully merged into the `master` branch. Therefore, our project could not simply fork Saros on GitHub but had to integrate the `raimondas2` branch from Gerrit, which was still actively developed at the time. As I also wanted to get a first assessment of Björn Kahlert's browser, I included it with minor modifications in our repository, too.
Now we had to test and – as it turned out – debug our integrated prototype under Windows, GNU/Linux, and Mac OS. Since we had no constant access to Mac OS, I extended our repository configuration by adding two branches `easy_testing` and `mac_testing` to ease the debugging by enabling the quick deployment of the current version on every operating system. These branches used compiled JAR files of Saros and the browser widget instead of the source code in order to eliminate the need to configure, update, and manage the integrated Git repositories on each test system. This accelerated the set-up immensely because most of our test systems were relatively slow virtual machines. `Mac_testing` additionally contained the special adjustments for Mac OS, which also had to be tested under Windows and GNU/Linux so that they would not break anything. The nice thing about this whole set-up was that the branches and therefore the entire project configuration could be switched seamlessly

by checking out another branch. No additional changes were required as all relevant IDE files were managed by Git. The `master` branch with the editable source code of Saros and Björn Kahlert's browser was required whenever the integration itself had to be adapted. After the appropriate changes had been made, the JAR files were recompiled and added to the branches so that the other systems could be tested.

5.1.2 Git documentation

As I noticed lacking knowledge of Git amongst new Saros members I documented the things I learned concerning Git and Gerrit in form of some recipes on the Saros homepage[1]. The covered topics were inspired by the questions I had been asked (more than once).

5.1.3 Documentation for new developers

Triggered by a developer test executed by Matthias Bohnstedt that found that new developers needed some orientation when extending the new HTML GUI, I wrote a detailed developer guide. This guide gives a template for adding new functionality to the UI layer and highlights the key Java classes. I published this both on the Saros homepage[2] and in source code to be accessed via the JTourbus[3].

Since the browser library might be adapted in the future, I highlighted the places to start, for example when switching the underlying browser[4].

5.1.4 Release process

I volunteered as test manager for the release process[5]. This basically meant that Arndt Lasarzik, the assistant test manager, and I verified the changelog that consisted of the fixes, features, and other changes in the span of more than one year.

5.1.5 Review process

As I repeatedly noticed that Saros members shied from submitting their patches to the review system, I co-initiated a discussion in the Saros team about problems and concrete proposals to improve the review process. This discussion was very productive and, thanks to the initiative of Franz Zieris, an update to the review process has been decided on and published.

[1] `https://www.saros-project.org/git-recipes` (retrieved 12 April 2015)
[2] `https://www.saros-project.org/html-gui` (retrieved 12 April 2015)
[3] `https://github.com/ag-se/jtourbus` (retrieved 31 March 2015)
[4] `https://github.com/ag-se/swt-browser-improved/wiki` (retrieved 12 April 2015)
[5] `http://www.saros-project.org/ReleaseProcess` (retrieved 20 March 2015)

5.2 Suggestions for improvement

5.2.1 Developing Javascript code

I will shortly illustrate a general bump in the workflow when developing Javascript to be executed from Java. The Javascript expressions are often one-liners. The problem is that the embedded browser has no debugging support for Javascript. A primitive alternative would be to use `alert` statements or other forms of logging, which is rather cumbersome. Furthermore, there is no IDE support such as syntax checks for Javascript inside Java Strings.
I do not have a perfect solution for this, but the way to go seems to be to develop the website in a way such that it can be opened outside of Saros in a browser. If necessary, state can be faked by static content. Then one can develop the Javascript using standard Javascript tools and debugging facilities. As soon as the code works, it may be copied into Java strings or included as methods in external Javascript files.

5.2.2 Continuous integration

The Jenkins job building a Gerrit patchset and the job building the Saros `master` branch should be the same or at least use the same configuration so that a successful build in Gerrit will not break when submitted into the master. This happened twice during my thesis when the Jenkins configurations were adapted. Since both jobs are structurally different, it is not only difficult to keep the configurations synchronised but also hard to use the same Ant scripts; currently, the `core` and `ui` module are built twice (see subsection 4.5.3).
Ideally, the Ant build (at least the compile tasks) should also be executable in the local development environment, especially for adapting the build scripts. Currently, this only works after manually rebuilding the Jenkins environment, i. e. installing all the used tools and libraries and setting the corresponding parameters. Even then, the classpaths constructed inside the Ant files are too long to be executed on Windows.
Lastly, the release process[6] requires a lot of manual steps and since the actual build is done on a local machine, it is difficult to reproduce. It depends on the installed bundles of all dependent libraries[7] in the local Eclipse installation, for example[8]. After I changed the instructions to use `git clean` in addition to `git reset --hard`, at least the local Git state should have no effect on the outcome anymore.

5.2.3 Release testing

Testing the new features and fixes for the Saros release 14.10.31 took a considerable amount of time. The reason for this was the long list of changes as the last release

[6] Described at `http://www.saros-project.org/ReleaseProcess#How_to_create_a_new_release` (retrieved 1 April 2015)
[7] One can define exact version numbers for these dependencies but that is laborious and currently not done.
[8] Out of interest I tried it on my machine and was unable to create a release following the guide.

had been more than a year ago. Furthermore, all the changes had been done before I became a part of the Saros team, so I was missing the necessary context. Many of the people who were involved in those changes had either left Saros or did not have the time for the release testing.

Even though the commit messages and bug tracker entries (if they existed) were not badly written, they did not contain enough information for newcomers to verify the changes. I would suggest the consequent creation of (GUI) tests for all fixes and features which are relevant for the changelog. For changes that are difficult to test automatically such as changes to the layout, bug tracker entries containing pictures and detailed steps should be written.

As it also has to be verified that the previous release exhibits a different behaviour, automatic tests are particularly convenient. Of course, they cannot replace all manual tests and the user acceptance tests will still be an integral part of the release process. The hope is that they can drastically reduce the two full days Arndt Lasarzik and I spent gathering the necessary information and then testing very mechanically.

5.3 Retrospective thoughts

I (in agreement with other Saros members) have made two major decisions in my thesis:

- To use the SWT browser and make it work.
- To develop Björn Kahlert's browser as a standalone library as opposed to creating a new wrapper and only copying the needed parts.

I will now give my thoughts about theses decisions in hindsight.

5.3.1 Was the SWT browser the right choice?

The potential risks of the SWT browser were apparent from the beginning, but it was unclear how well they could be eliminated or whether they would become concrete problems. After the initial embedding problems into IntelliJ had been solved, the SWT browser presented a promising candidate on most systems. Even though it seemed like an uphill battle at times, I think the considerable initial effort paid off.

The SWT browser was the first logical step to gather more information about a web-based GUI and especially the resulting source code because – in spite of all the challenges – it was still the only browser that could be directly integrated into Saros. The JxBrowser might have been an alternative but I consider it to be unlikely that users will accept a 200 MB library and it has similar problems when embedded into Eclipse. For JavaFX, Saros would first have to drop Java 6 support and thus exclude IntelliJ on Mac OS.

In conclusion, I am positive that starting with the SWT browser was the right choice. After having learnt more about the use of a browser for IDE plug-ins, Saros can still decide to switch to another one.

5.3.2 What about the decision to use Björn Kahlert's browser?

There were two options: First, writing an own browser extension and copying just the needed parts from Björn Kahlert's implementation and, second, using his browser as a whole and making the necessary adaptations. I have chosen the second alternative (for the reasons see section 2.2.3). Even though my modifications were considerably larger than originally thought, I still think that this decision was valid. The main reasons for the higher effort were, on the one hand, that some required internal changes only became apparent after a close look and, on the other hand, that the target environment had special needs and was not as stable as initially assumed (see section 4.2.2). The second category would also have affected a fresh implementation, possibly at different locations in the code.
I am still convinced that the fundamental complexity of a re-implementation would be similar to that of Björn Kahlert's solution (see section 2.2.3). However, as it is not certain how many of the optional features of the browser wrapper will be used, it is too soon to make a final assessment.

5.3.3 Change of focus throughout my thesis

Just before I registered my Master's thesis, the implementation of the HTML-GUI became relevant (again), as Damla Durmaz had successfully implemented her activity log inside a browser on the second try. Up to this point, my focus was to implement STF or an equivalent test framework for IntelliJ such that the same tests could be used for both IDEs.
In the end, I started out with the objective to develop a GUI test framework for whatever GUI technology would be favoured after the evaluation. Because no evaluation of HTML-based approach had been done yet, I was asked to join forces with Matthias Bohnstedt for this.
Due to personal time conflicts he had to postpone the start of his thesis, so the evaluation became a big part of my work. In addition, the evaluation itself turned out to be much more time-intensive than originally thought since more and more technical problems arose that had to be fixed[9].
Even after the general approach, using HTML, was decided on, the GUI first had to be implemented as a prototype before any GUI tests would make sense. Therefore, I did this part too and even integrated Björn Kahlert's browser component to provide a convenient and reliable browser API for Saros. Finally, I found time to concern myself with the original aim, creating GUI tests. Since the HTML-GUI still only features a few functions, I merely sketched what the interface of a test framework might look like and implemented the first actual GUI tests for both IDEs.

[9] If I had not been able to fix them, the evaluation result would have been a lot clearer a lot sooner.

6 Results of this thesis

6.1 Conclusion

My thesis has laid the necessary groundwork for a complete implementation of Saros using the HTML-based approach. After I had evaluated this approach to be a promising way to eliminate redundant GUI code, I identified the SWT browser as the first (temporal) choice for a complete implementation (section 2.4). I have given an extensive overview on Java web browsers including their weaknesses and investigated their applicability for Saros (section 2.2).

The objective of the prototype implementation is to cover all the different facets of a browser-based IDE plug-in implementation (section 4.1). This includes embedding the browser into Eclipse and IntelliJ on GNU/Linux, Windows, and Mac OS X (section 3.1), creating an HTML sketch of all the relevant GUI elements, and connecting the HTML-GUI to the Saros business logic. Connecting it to the Saros logic means both displaying the application state in the browser (subsection 4.1.3) and sending the user input to the Saros backend(subsection 4.1.1 and subsection 4.1.2). The mechanics of GUI tests have also been covered and are based on the existing Eclipse GUI test framework (STF) with sensible adaptations for HTML (section 4.3). The resulting prototype is already integrated into the complete Saros build and development infrastructure (section 4.5) such that each developer can easily activate it via a feature toggle and contribute to it.

During this implementation, I had a close look at the communication between Java and Javascript with regard to general inconveniences that are caused by the mix of those two languages, e. g. missing IDE support and compile-time checks (subsection 4.1.2 and subsection 5.2.1). Simplifying the development for future developers, especially in terms of threading and asynchronous execution, was my focus while I was extending the browser (section 4.2). This extension was originally created by Björn Kahlert and I subsequently developed it into a generic, standalone library. The original motivation to extend the browser in the first place was to make Javascript commands reliable by possibly delaying the execution until a defined state is reached.

Since the SWT browser has compatibility problems under GNU/Linux and Mac OS X (section 2.2.2), the library has been prepared for a switch of the underlying browser (section 4.2.7). The most promising alternative for the future is supposed to be the JavaFX browser which, however, requires at least Java 7. The JxBrowser could also be an immediate alternative for IntelliJ on certain operating systems.

6.2 Future work

Although the foundation for a future HTML-GUI has been laid, there are still many steps to go. The technologies used on the HTML and Javascript side have not been evaluated and therefore all of my HTML code is only temporary. Bastian Sieker [Sie15] will work on that topic in his Master's thesis.
I have only conducted a basic evaluation of the resulting code and the accompanying development process. As soon as the code has grown and more GUI elements have been implemented, a more detailed comparison between the traditional SWT development and the new HTML one makes sense; currently the choice of Javascript framework has not been made. Matthias Bohnstedt's Master's thesis [Boh15] is going in this direction.
The scope of the current HTML GUI being minimal is, among others, a result of the lack of business functionality in the Saros `core` module. Large parts of the business logic are still in the Eclipse and IntelliJ specific modules.
As already mentioned, the choice of the used browser is not final. Currently, IntelliJ started with Java 7 or 8 is unsupported on Mac OS X. A good time to re-evaluate the choice of browser would be when Saros drops support for Java 6[1]. In this context a more complete compatibility matrix should be created with special focus on different SWT and Eclipse versions. I have mainly concentrated on the basic support on all operating systems across Java versions. The related decision of which SWT version to use for compilation and which to ship for each operating system has not been made (currently it is 3.6).
While the problems of the SWT browser have been identified and it has been thoroughly tested, the same cannot be said for the alternative, the JavaFX browser. Although it wins on paper, only its real application inside an IDE plugin is able to reveal its quirks, especially since it is a relatively new technology.
Further points for future development are an expansion of the GUI test set, the handling of GUI elements outside of the Saros view in both IDEs, and changing Saros' automatic test systems to include more different operating systems.

[1] I have documented the places where the changes have to be made in `https://github.com/ag-se/swt-browser-improved/wiki` (retrieved 12 April 2015).

Bibliography

[Ben15] Sabine Bender. Working title: Evaluation einer Portierung von Saros auf Netbeans. Bachelor's thesis, Freie Universität Berlin, 2015.

[Blo08] Joshua Bloch. *Effective Java Second Edition*. Addison Wesley, Boston, MA, USA, 2008.

[Boh15] Matthias Bohnstedt. Working title: Cross-Platform-GUI Entwicklung in Saros. Master's thesis, Freie Universität Berlin, 2015.

[Che11] Lin Chen. Einführung eines Testprozesses. Diploma thesis, Freie Universität Berlin, 2011.

[Dje06] Riad Djemili. Entwicklung einer Eclipse-Erweiterung zur Realisierung und Protokollierung verteilter Paarprogrammierung. Diploma thesis, Freie Universität Berlin, 2006.

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

[Eva03] Eric Evans. *Domain-Driven Design: Tacking Complexity in the Heart of Software*. Addison-Wesley, Boston, MA, USA, 2003.

[GBB+06] Brian Goetz, Joshua Bloch, Joseph Bowbeer, Doug Lea, David Holmes, and Tim Peierls. *Java Concurrency in Practice*. Addison-Wesley Longman, Amsterdam, 2006.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, Boston, MA, 1995.

[Kah15] Björn Kahlert. *Usability der auf Templatemetaprogrammierung basierenden Softwarebibliothek "SeqAn"*. PhD thesis, Freie Universität Berlin, 2015.

[Las15] Arndt Lasarzik. Working title: Refaktorisierung des Eclipse Plugins Saros für die Portierung auf andere IDEs. Bachelor's thesis, Freie Universität Berlin, 2015.

[Lea00] Douglas Lea. *Concurrent programming in Java: design principles and patterns*. Addison-Wesley Professional, 2000.

[LMS+12] Fred Long, Dhruv Mohindra, Robert C. Seacord, Dean F. Sutherland, and David Svoboda. *The CERT Oracle secure coding standard for Java.* The SEI series in software engineering. Addison-Wesley, Upper Saddle River, NJ, 2012.

[Ros11] Stefan Rossbach. Einführung einer kontinuierlichen Integrationsumgebung und Verbeserung des Test-Frameworks. Bachelor's thesis, Freie Universität Berlin, 2011.

[Sie15] Bastian Sieker. Working title: User-Centered Development of a JavaScript and HTML-based GUI for Saros. Master's thesis, Universität Paderborn, 2015.

[Szü10] Sandor Szücs. Behandlung von Netzwerk- und Sicherheitsaspekten in einem Werkzeug zur verteilten Paarprogrammierung. Diploma thesis, Freie Universität Berlin, 2010.