**Freie Universität Berlin**

Bachelorarbeit am Institut für Informatik der Freien Universität Berlin

Arbeitsgruppe Software Engineering

# Überbrückung der Sprachgrenze zwischen Java und JavaScript in der GUI des Saros-Plug-ins

Daniel Paul-Gattringer
Matrikelnummer: 4870892
danielpaul@zedat.fu-berlin.de

Betreuer: Kelvin Glaß
Eingereicht bei: Prof. Dr. Lutz Prechelt
Zweitgutachter: Prof. Dr.-Ing. Jochen Schiller

Berlin, 20. Dezember 2018

**Zusammenfassung**

Saros ist als ein Plug-in für die IDE Eclipse vorgesehen, um Entwicklern die verteilte Paarprogrammierung zu ermöglichen. Um dieses Plug-in auch für andere Entwicklungsumgebungen bereit zustellen, ohne separate GUIs zu entwickeln, evaluierte und implementierte Christian Cikryt den Prototypen einer HTML-GUI. Dieser wurde in weiteren Arbeiten optimiert. Bei der Implementierung der HTML-GUI entstanden mehrere Klassen und Funktionsrümpfe, die sowohl in Java als auch in JavaScript implementiert werden mussten.
Ziel dieser Arbeit ist die Schnittstelle zwischen der HTML-GUI von Saros und dem Saros-Core mit einer geeigneten Technologie zu überbrücken. Um dieses umsetzen zu können, stelle ich zuerst Anforderungen auf. Die Anforderungen ergeben sich aus der Problemstellung, dem Build-Prozess sowie dem aktuellen Quellcode von Saros. Anschließend evaluiere ich die Technologien nach den aufgestellten Kriterien. Dabei stellt sich Kotlin als Lösung heraus.
Abschließend implementiere ich mit dieser Technologie das UI-Modul neu. Während der Implementierung stellt sich heraus, dass es lediglich vier Klassen gibt, die sowohl in Java als auch in JavaScript verwendet werden. Durch die zukünftige Weiterentwicklung der HTML-GUI gewinnt diese Implementierung mit Kotlin jedoch immer weiter an Wert.

## Eidesstattliche Erklärung

Ich versichere hiermit an Eides Statt, dass diese Arbeit von niemand anderem als meiner Person verfasst worden ist. Alle verwendeten Hilfsmittel wie Berichte, Bücher, Internetseiten oder ähnliches sind im Literaturverzeichnis angegeben, Zitate aus fremden Arbeiten sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

20. Dezember 2018

Daniel Paul-Gattringer

# Inhaltsverzeichnis

# 1 Einführung

## 1.1 Paarprogrammierung

Bei der Paarprogrammierung handelt es sich um eine Arbeitsweise aus dem Extreme Programming. Hierbei arbeiten zwei Entwickler an einem Computer und erarbeiten gemeinsam eine Lösung für ein Problem. Während einer der beiden die Lösung implementiert, prüft der andere zeitgleich die Korrektheit. Diese Methode Software zu entwickeln, bringt mehrere Vorteile mit sich. Es können durch das Vier-Augen-Prinzip Fehler vermieden werden, da das Problem und die Lösung gemeinsam besprochen werden. Durch die Diskussion der Lösung kann zum Einen bessere Software entstehen, als auch ein Wissensaustausch zwischen den Beteiligten stattfinden. Die Paarprogrammierung bringt jedoch auch Nachteile mit sich. Die Geschwindigkeit der Entwicklung kann durch fortwährende Dialoge gehemmt werden. Des Weiteren gibt es einige Entwickler, die effizienter allein arbeiten. Den größten Nachteil bildet der gemeinsame Aufenthaltsort der Entwickler, um die Lösung gemeinsam besprechen und erarbeiten zu können. An dieser Schwäche setzt die verteilte Paarprogrammierung an. Diese ermöglicht, mithilfe einer entsprechenden Software wie zum Beispiel Saros, den Entwicklern an unterschiedlichen Orten zusammen an einer Lösung zu arbeiten.

## 1.2 Was ist Saros?

Saros ist ein Plug-in, welches an der Freien Universität Berlin ursprünglich für die Entwicklungsumgebung Eclipse programmiert wurde. Es ermöglicht in einer sogenannten Session oder auch Sitzung, dass bis zu 5 Entwickler zeitgleich an einer Lösung arbeiten können. Um eine Sitzung zu starten wird ein XMPP-Account benötigt.

> **XMPP**
> XMPP ist die Abkürzung für Extensible Messaging and Presence Protocol, dessen Quellcode öffentlich einsehbar ist. Es ist eine Sammlung von Technologien für Chats bzw. Instant-Messaging-Diensten, Anwesenheitsanzeigen, Gruppenchats und einigen mehr. Entwickelt wurde es in der Jabba-Community.

Mit diesem Account wird sich eingeloggt und es kann dem Partner oder den Partnern über die Kontaktliste eine Einladung zu einer Sitzung zugesendet werden. Sobald diese angenommen wurde, werden die ausgewählten Projekte und Dateien in Echtzeit geteilt. Alle Beteiligten können nun Änderungen vornehmen und verfolgen. Um die Kommunikation von Lösungswegen zu erleichtern, stehen den Teilnehmern ein Chat und ein Whiteboard zur Verfügung. Bei dem Whiteboard handelt es sich um eine separate Ansicht. Diese besteht aus einer weißen Fläche, die zum Skizzieren der Lösung verwendet werden kann. Diese Ansicht wird mit den Teilnehmern einer Sitzung geteilt.

## 1.3 Motivation

Saros war ursprünglich für die IDE Eclipse entwickelt worden und besteht aus mehreren Modulen unter anderem dem Saros-Core und der Saros-UI. Im Core sind al-

le Funktionen zusammengefasst, die unabhängig von der IDE benötigt werden, wie zum Beispiel das Monitoring der Dateien oder die Verbindung zwischen den Teilnehmern. Um Saros ebenfalls für die Entwicklungsumgebung IntelliJ IDEA bereitzustellen, musste eine separate grafische Oberfläche erstellt werden. Der Grund hierfür bestand im technischen Aufbau von IntelliJ IDEA, diese verwendet ein anderes GUI-Toolkit zum Aufbau der grafischen Oberfläche als Eclipse. Vorerst wurden für beide Entwicklungsumgebungen unterschiedliche GUIs implementiert. Dies führte unweigerlich zu einem Mehraufwand an Quellcode und erhöhte die Fehleranfälligkeit, da Änderungen immer in beiden GUIs erfolgen mussten. Um dies zu verhindern, hat Christian Cikryt 2015 [1] in seiner Masterarbeit den Einsatz eines Browser als GUI evaluiert und mithilfe von HTML, CSS und JavaScript die Grundlagen bereit gestellt. Eine browsergestütze GUI kann in beide IDEs eingebettet werden. Bastian Sieker optimierte mit seiner Masterarbeit [5] diesen Prototypen, jedoch fehlt weiterhin der Großteil an Funktionalität. Durch den kombinierten Einsatz von Java und JavaScript ist der Programmcode schwierig statisch zu überprüfen und die Einarbeitung in den Quellcode wird erschwert. Des Weiteren müssen einige Klassen mit identischen Funktionen sowohl in Java als auch in JavaScript implementiert werden, die den Quellcode ohne weiteren Nutzen vergrößern.

### 1.4 Ziel dieser Arbeit

Mit meiner Bachelorarbeit ziele ich darauf ab, die Schnittstelle zwischen Saros-HTML-GUI und Saros-Core mithilfe einer Technologie zu überbrücken. Mit dieser Lösung soll es möglich sein, die mehrfache Implementierung von Klassen zu beheben. Damit kann ich nicht nur die Größe des Quellcodes, sondern auch die Fehleranfälligkeit verringern. Mit der neuen Technologie möchte ich zukünftigen Entwicklern ermöglichen, sich leichter in den Quellcode einzuarbeiten. Durch die unklare Struktur zwischen Java und JavaScript wird dies aktuell erschwert. Um dieses Ziel erfolgreich umzusetzen, arbeite ich mich zuerst in das aktuelle Saros-UI-Modul ein. Daraufhin untersuche ich, welche Technologien in Frage kommen und welche davon am besten dafür geeignet ist. Im Anschluss arbeite ich mich in die geeignete Technologie ein, um abschließend den Programmcode und die dazugehörigen Tests zu implementieren. Mit dieser Bachelorarbeit möchte ich lediglich Änderungen an der Bedienschnittstelle zwischen dem Saros-Core und der HTML-GUI oder vielmehr dem UI-Modul vornehmen.

## 2 Evaluierung der Technologien

In diesem Abschnitt befasse ich mich mit den Anforderungen an die zu verwendende Technologie. Anschließend nenne und beschreibe ich die Möglichkeiten und erläutere, warum ich mich für Kotlin entschieden habe.

### 2.1 Anforderungen

Im Kapitel 1.3 und 1.4 stelle ich das Problem und das Ziel dieser Arbeit vor. Es lassen sich aus diesen Kapiteln bereits die folgenden funktionalen Anforderungen ableiten. Die Technologie produziert JVM-Bytecode oder ist kompatibel mit Java. Weiter-

hin muss der Programmcode sich in JavaScript kompilieren lassen. Aus dem Build-Prozess kann ich eine weitere, funktionale Anforderung ableiten. Für das Saros-Build wird aktuell das Build-Werkzeug Ant verwendet, jedoch soll Gradle als Nachfolger genutzt werden. Somit muss ich die zukünftige Software sowohl in den Build-Prozess von Ant als auch in den von Gradle integrieren können.
Wie bereits in den Arbeiten von Nina Weber [7] und Marius Schidlack [4] angemerkt, soll die neue Technologie im optimalen Fall die Einarbeitung in den Quellcode verbessern. Dies füge ich ebenfalls zur Liste der Anforderungen hinzu.

Zusammengefasst ergeben sich folgende Anforderungen

- funktionale Anforderungen
  1. Produziert JVM-Bytecode oder ist Java kompatibel
  2. Programmcode lässt sich in JavaScript kompilieren
  3. Integration in den aktuellen Build-Prozess von Ant- bzw. Gradle
- nicht funktionale Anforderung
  - Einarbeitung in den Quellcode erleichtern

## 2.2 Die Suche nach Technologien

Mein Betreuer machte mich mit Scala und dem dazugehörigen Framework Scala.js bereits auf die erste Technologie aufmerksam. Mit diesem Anlaufpunkt beginne ich meine Suche, verschaffe mir einen kurzen Überblick über das Framework[1] und die Programmiersprache[2] und setze meine Suche nach weiteren Technologien fort. Dazu nutze ich eine Übersicht [6] derzeitiger Programmiersprachen, welche die JVM verwenden. Dabei bin ich auf zwei weitere Vertreter gestoßen Fantom und Kotlin, da diese sowohl Java-Bytecode als auch JavaScript kompilieren können.

### 2.2.1 Scala und Scala.js

Scala wurde von 2001 bis 2004 im schweizer Institue École polytechnique fédérale de Lausanne entwickelt. Sie ist C# und auch Java sehr ähnlich und vereint Konzepte objekt-orientierter und funktionaler Programmiersprachen. Für Scala bedeutet das ”jeder Wert ist ein Objekt und jede Operation ein Methodenaufruf ” [3]. Scala erweitert den Funktionsumfang von Java immens, indem es nicht nur funktionale Eigenschaften hinzufügt, sondern ermöglicht eigene Operatoren hinzuzufügen. Um jedoch auch JavaScript-Code zu erzeugen, wird Scala.js als zusätzliches Framework benötigt. Es erweitert die Anzahl der verwendeten Frameworks erneut. Dies sehe ich als negativen Aspekt an, da jedes zusätzliche Framework den Wartungsaufwand erhöht. Des Weitere können durch den Einsatz mehrerer Frameworks negative Seiteneffekte entstehen.

[1] https://www.scala-js.org/
[2] https://www.scala-lang.org/

**Framework**
Ein Framework in der Softwareentwicklung ist ein Grundgerüst, welches dem Entwickler zwar die Architektur seiner Software vorgibt, ihm jedoch mit Bausteinen unterschiedlicher Art die Entwicklung erleichtert. Zumeist werden Klassen und Bibliotheken mitgeliefert, ebenso wie unterschiedliche Komponenten für die Laufzeitumgebung.

Bei der Einarbeitung in die Sprache stellte ich fest, dass diese ausführlich und mit einigen Beispielen dokumentiert ist. Um Scala kompilieren zu können, wird der Compiler scalac benötigt. Diesen in den Buildprozess, der zum damaligen Zeitpunkt mit Ant umgesetzt wurde, zu integrieren, würde einen zusätzlichen Aufwand bedeuten. Die Integration in den Gradle-Prozess ist durch mindestens ein Plug-in[3] möglich. Dafür bietet Scala, durch die Eigenschaften einer funktionalen Programmiersprache, die Möglichkeit den aktuellen Quellcode an vielen Stellen zu kürzen. Um dabei nicht an Verständlichkeit zu verlieren, setzt es Wissen in mindestens einer funktionalen Programmiersprache wie zum Beispiel Haskell voraus.

### 2.2.2 Fantom

Als Nächstes stieß ich auf Fantom. Eine durch Andy und Brian Frank entwickelte Sprache[4], die erstmals 2005 veröffentlicht wurde und seither stetig weiter entwickelt wird. Das Ziel dieser Sprache ist es, eine allgemeine Standardschnittstelle zwischen Java, JavaScript und der Dotnet bzw. Common Runtime Language zu bilden. Sie ermöglicht es Code zu implementieren, der in der JVM, JavaScript oder in C# verwendet werden kann. Dies wird durch entsprechende Skripte ermöglicht.
Damit erfüllt Fantom bereits die ersten beiden funktionalen Anforderungen. Durch die starke Nähe zu Java und C# ist die Hürde eines Wechsels zu Fantom sehr gering. Beim Versuch die ersten kleinen Schritte mit Fantom zu machen, fiel mir jedoch auf, dass es zwar separate Entwicklungsumgebungen für Fantom gibt, jedoch weder für Eclipse noch für IntelliJ IDEA stehen aktuelle Plug-ins zur Verfügung. Das für IntelliJ IDEA bereitgestellte Plug-in[5] ist veraltet und lediglich in der Version 0.0.6 von 2009 verfügbar. Das Arbeiten mit zwei IDEs erschwert die Einarbeitung, da der Wechsel von einer IDE zur anderen den Arbeitsprozess verlangsamt und die Fehleranfälligkeit erhöht. Dies widerspricht der Anforderung aus 2.1. Weiterhin empfinde ich den Build-Prozess im Zusammenhang mit dem Saros-Plug-in als eine Herausforderung. Hier müsste ich mit eigenen Skripten aufwendig den Build-Prozess anpassen.

### 2.2.3 Kotlin

Als dritte Möglichkeit stieß ich auf Kotlin. Diese von JetBrains entwickelte Programmiersprache vereint die Konzepte objektorientierter und funktionaler Programmiersprachen. Des Weiteren bietet diese eine gute Unterstützung von Entwicklungsumgebungen wie IntelliJ IDEA oder Eclipse. Ähnlich wie Fantom, ist Kotlin sehr stark an

[3] https://github.com/gtache/scalajs-gradle
[4] https://fantom.org/
[5] https://plugins.jetbrains.com/plugin/4241-fantom-support

Java angelehnt und bietet dadurch eine sehr gute Kompatibilität. Java-Klassen kann ich ohne große Umschweife direkt verwenden und umgekehrt. Auch die Einarbeitung in die Programmiersprache stellt keine große Herausforderung dar. Die Sprache ist sehr gut dokumentiert und da Java-Klassen direkt unterstützt werden, können diese vorerst beibehalten werden. Kotlin wird in Eclipse und in IntelliJ IDEA jeweils durch ein eigenes Plug-in unterstützt. Dies kann bei einer Installation angewählt oder bei einer bestehenden Installationen ohne Umschweife nachträglich installiert werden.
In Kotlin implementierte Klassen, aber auch einzelne Methoden können ebenfalls zu JavaScript-Code kompiliert werden. Hierfür wird im Build-Prozess, aktuell in Ant später auch in Gradle, ein Plug-in und ein weiterer Arbeitsschritt eingefügt. Mit diesem Arbeitsschritt werden die vorher angegebenen Klassen zu JavaScript-Code kompiliert. Zu beachten ist hier jedoch, dass lediglich der Kotlincode kompiliert wird. Jegliche in den zu kompilierenden Klassen verwendete Java- oder Kotlinbibliothek wird ignoriert. Dies kann in der von den Entwicklern bereitgestellten Dokumentation[6] nachgelesen werden. Ich möchte als weiteren Vorteil erwähnen, dass Kotlin direkt mit React[7] verwendet werden kann. Dies kann als Vorteil angesehen werden, da das UI.Frontend-Modul mit diesem Framework implementiert wurde. Des Weiteren ist die Verwendung von Kotlin im Zusammenhang mit React, wie die Integration in den Build-Prozess von Gradle, sehr gut dokumentiert.

## 2.3 Zusammenfassung der Evaluation

| | Fantom | Scala | Kotlin |
|---|---|---|---|
| Java kompatibel | ++ | ++ | ++ |
| Ausgabe von JavaScriptcode | + | +/- | ++ |
| Integration in Build-Prozess | – | + | ++ |
| Dokumentation/Einarbeitung | +/- | + | ++ |

Tabelle 1: Die Auswertung im Bezug auf die Anforderungen

Fantom schloss ich trotz der starken Nähe zu Java und der dadurch verkürzten Einarbeitungszeit aufgrund der kleinen Community, des aufwendigen Build-Prozesses und dem aktuellen Status "in der Entwicklung" für den weiteren Verwendungszweck aus.

Scala bietet einen sehr großen Funktionsumfang. Auch die Nähe zu Java ist ein Punkt, der für Scala spricht. Andererseits wird neben Scala noch ein zusätzliches Framework, in die bereits vorhandene Vielfalt an Werkzeugen, hinzugefügt. Einen weiteren negativen Aspekt sehe ich im zusätzlichen Aufwand, der benötigt wird, um Scala und Scala.js in den aktuellen und auch zukünftigen Build-Prozess einzubauen. Aus diesem Grund habe ich auch diese Programmiersprache ausgeschlossen.

Daraus ergibt sich für mich, dass Kotlin, nach aktuellem Stand, für diese Problemstellung die optimale Lösung ist. Diese Programmiersprache bietet die nötige Kompatibilität zu Java, ermöglicht mit wenig Aufwand das Kompilieren zu JavaScript und

[6] https://kotlinlang.org/docs/reference/kotlin-doc.html
[7] https://reactjs.org/

lässt sich mit ebenso wenig Aufwand in den aktuellen Build-Prozess in Ant, wie auch später in den von Gradle integrieren.

# 3 Implementierung

In diesem Abschnitt befasse ich mich erst mit dem Aufbau von Saros, speziell mit den von mir bearbeiteten Modulen. Anschließend stelle ich Kotlin kurz mit Beispielen aus den Arbeitsergebnissen vor und gehe auf Probleme und Besonderheiten ein, die während der Implementierung aufgetreten sind.

## 3.1 Aufbau von Saros

Saros besteht aus mehreren Modulen. Auf die für diese Arbeit wichtigsten Module möchte ich kurz eingehen. Das Core-Modul enthält alle grundlegenden Funktionen, die das Eclipse- und das IntelliJ IDEA-Plug-in sowie der eigenständige Saros Server benötigen. Es stellt die Funktionen zum Verbindungsaufbau, zum Monitoring von Dateien und zur Kommunikation bereit. Das UI.Frontend-Modul enthält den gesamten Quellcode der HTML-GUI. Dieses Modul ist mit dem Framework React entwickelt worden. Mithilfe von NPM werden Abhängigkeiten aufgelöst, die notwendige Software installiert und anschließend wird der Quellcode in Form von jsx-Dateien zu HTML, JavaScript und CSS kompiliert.

> **NPM** ist ein Paketmanager für JavaScript und stellt über eine Registrierung unterschiedliche Software bereit. Es ermöglicht Code zu installieren, zu teilen und zu verteilen.

Das UI-Modul bildet die Schnittstelle zwischen Frontend- und Core-Modul. Es enthält folgende Pakete:

- dpp
- dpp.ui
- dpp.ui.browser_functions
- dpp.ui.core_facades
- dpp.ui.ide_embedding
- dpp.ui.manager
- dpp.ui.model
- dpp.ui.pages
- dpp.ui.renderer
- dpp.ui.util

Das *dpp*-Paket enthält die *HTMLUIContextFactory*. Diese Fabrik erstellt mithilfe des Frameworks PicoContainer alle Objekte, die plattformunabhängig benötigt werden und hält diese in einem Container vor.

**PicoContainer**
PicoContainer ist ein Framework, das von Aslak Hellesoy und Jon Tirsen entwickelt wurde. Es unterstützt den Entwickler, indem das Framework ein beliebiges *java.lang.class* Objekt übergeben wird und es eine Instanz dieser Klasse zurückgibt oder in einem Container vorhält. Implementiert diese Klasse die Schnittstelle *Startable()*, werden mit der Methode *start()* gegebenenfalls Objekte erzeugt, die bei der Erstellung benötigt werden. PicoContainer ist eine Möglichkeit, Dependency Injection umzusetzen.

**Dependency Injection**
Dependency Injection[2] ist ein Entwurfsmuster in der objektorientierten Entwicklung, dass die Abhängigkeiten eines Objektes zur Laufzeit regelt. Zum Beispiel benötigt ein Objekt zur Initialisierung ein anderes Objekt, erstellt es das anhängige Objekt nicht selbst, sondern erhält es aus einem zentralen Ablageort. Es gibt unterschiedliche Arten von Dependency Injection unter anderem Contructor Injection, Setter-Injection und Interface-Injection.

Des Weiteren beinhaltet das Paket das Enum *HTMLUIStrings*, welches unter anderem alle Titel und Label enthält, die für das UI-Paket benötigt werden.
Im *dpp.ui*-Paket sind die Views als Enum hinterlegt. Die wichtigere Klasse ist jedoch die *JavaScriptAPI*. Dies ist die Schnittstelle, die die Kommunikation von Java zu JavaScript ermöglicht. Sie beinhaltet Funktionen zur Aktualisierung des Accounts, des Status und des Projektbaums. Weiterhin enthält es Funktionen, um Ereignisse und Fehler an JavaScript zu übergeben.
Die Kommunikation von JavaScript zu Java ist im Paket *browser_functions* implementiert. Hier wird jede Browserfunktion, die über die HTML-GUI zur Verfügung stehen soll, programmiert. Diese können dann in JavaScript mithilfe von Invoking aufgerufen werden.

**Invoking**
Invoking ist eine Methode um Java-Methoden innerhalb von JavaScript aufzurufen. Diese Methoden werden über den Paketnamen identifiziert. Sie können je nach Programmierung im selben Thread (synchron) oder in einem separaten Thread (asynchron) ausgeführt werden.

Die grundlegenden Klassen, auf die alle weiteren im Paket basieren, sind die Klassen *SelfRegisteringJavascriptFunction*, *TypedJavascriptFunction* und *TypeRefinery*. Die *SelfRegisteringJavascriptFunction*-Klasse hat die Aufgabe, die Funktionen im Browser zu registrieren, das heißt sie werden dem PicoContainer zur Verfügung gestellt und Abhängigkeiten werden durch Dependency Injection aufgelöst. Die Klasse *TypedJavascriptFunction* filtert, die durch JavaScript erhaltenen, Aufrufargumente und wandelt diese in die zugehörigen Datentypen in Java um. Hierzu wird die Klasse *TypeRefinery* verwendet, die die notwendigen Methoden zum Umwandeln bereitstellt.
Die IDE-unabhängigen Klassen sind im *dpp.ui.ide_embedding* enthalten. Hier sind der

*BrowserCreator*, der *DialogManager*, das Interface *IBrowserDialog* und *IUIResourceLocator* implementiert. Der *DialogManager* stellt die grundlegenden Funktionen eines Dialoges bereit und verwaltet alle geöffneten Dialoge. Der BrowserCreator hält alle registrierten Browserfunktionen vor und übergibt diese via Depency Injection an den von der IDE abhängigen Browser. Er wird auch zum Erstellen einer Browser-Instanz benötigt. Das Paket *dpp.ui.manager* besteht aus dem *BrowserManager* und dem ProjectListManager. Erster verwaltet, die vom BrowserCreator erstellten, Browser-Instanzen. Hier sind hinzufügen, löschen und zurückgeben einer Browser-Instanz implementiert. Dem gegenüber ist der *ProjectListManager* verantwortlich für das Erstellen und Verwalten von ProjectTree-Instanzen. Diese Klasse ist auch für die Verlinkung mit den Ressourcen verantwortlich.
Die Modellklassen sind im Paket *dpp.ui.model* hinterlegt. Diese beinhalten *Contact*, *ProjectTree* mit der inneren Klasse *Node*, *State* und *ValidationResult*. Die genannten Klassen werden direkt oder indirekt auch im UI.Frontend-Modul verwendet, sind dort jedoch in JavaScript implementiert. Im Fall der *ProjectTree*-Klasse ist diese als JSON vorhanden.
Jede Browserseite benötigt auch ein Gegenstück in Java. Dieses ist in einem separaten Paket dpp.ui.pages implementiert. Hier werden, durch sogenannte Renderer, die Status der Seiten von Java in JavaScript übertragen und dort aktuell gehalten.
Im Paket *dpp.ui.renderer* sind die aus dem vorherigen Paket bekannten Renderer implementiert. Der Kern dieses Paketes ist die abstrakte Klasse *Renderer*, welche die grundlegenden Funktionen der anderen Renderer bereitstellt. Neben der Klasse *AccountRenderer*, gibt es noch die Klassen *StateRenderer*, *ProjectListRenderer* und *ContactRenderer*. Deren Aufgaben lassen sich aus dem Namen ableiten.

## 3.2 Kotlin

In diesem Kapitel gebe ich einen kurzen Einstieg in Kotlin und gehe auf die Besonderheiten dieser Sprache in Bezug auf die Reimplementierung des UI-Moduls ein. Hierzu gebe ich an den entsprechenden Stellen, Beispiele aus meinen Arbeitsergebnissen.

### 3.2.1 Einstieg in Kotlin

Kotlin ist nicht nur kompatibel mit Java, der Code in Kotlin ist darüber hinaus dem von Java ähnlich. Schauen wir uns dazu das Listing 1 an.

Listing 1: Contact

```
class Contact(val displayName: String?, val presence: String?, val
    addition: String?, val jid: String?)
```

Hier sind bereits die ersten Vorteile von Kotlin zu sehen. Das Schlüsselwort *class* ist aus Java bekannt. Anschließend folgt der Name der Klasse. In Java würde nun eine { und der Konstruktor sowie die Definition von Feldern, Eigenschaften und Methoden folgen. In Kotlin jedoch legen wir direkt nach dem Klassennamen bereits den primären Konstruktor mit den notwendigen Parametern fest. Im Beispiel sind das *displayName*,*presence*, *addition* und *jid*. Im Gegensatz zu Java wird bei Kotlin zuerst

der Parametername und dann der Parametertyp getrennt durch einen Doppelpunkt definiert. Das Schlüsselwort *val* ersetzt das aus Java bekannte *final*. Im Beispiel ist *val displayName* nicht nur ein Parameter, dadurch wird auch ein nicht veränderbares Feld erstellt. Diesem Feld wird direkt beim Erstellen des Objektes ein Wert zugewiesen. Für Java wird automatisch ein Getter erstellt. Ein veränderliches Feld hätten wir an dieser Stelle mit *var* angegeben. Für die Verwendung in Java stellt der Compiler automatisch neben einem Getter auch den Setter bereit. Im Gegensatz zu Java muss in Kotlin explizit angegeben werden, dass eine Variable auch den Wert *null* annehmen kann. Dies wird mit dem *?*-Symbol hinter dem Parametertype festgelegt. Ist dies nicht geschehen und man weist der Variable den Wert *null* zu, wird eine *NullPointerException* geworfen. Der Compiler unterstützt uns hierbei, in dem er bereits beim kompilieren eine *TypeCastException* anzeigt.
Die Definition von Methoden erfolgt in Kotlin anders als in Java. Schauen wir uns dazu das folgende Listing an.

Listing 2: toString

```
override fun toString() : String {
     return label + ''('' + type + ''}''
   }
```

In Kotlin werden Methoden mit *fun* und dem Methodennamen definiert. Die Parameter werden wie vom Konstruktor bekannt, mit Parameternamen und Parametertyp in Klammern angegeben. Erst zum Schluss wird der Typ des Rückgabewerts der Methode festgelegt. Muss eine geerbte Methode überschrieben werden, so geben wir dies mit dem Schlüsselwort *override* direkt zu Beginn der Definition, wie im Beispiel zu sehen ist. Wir können stattdessen auch die aus Java bekannte Annotation *@override* verwenden.
Während in Java jeder Befehl mit einem Semikolon abgeschlossen wird, verzichtet Kotlin darauf ganz. Wir könnten Semikolons verwenden, diese werden aber vom Compiler ignoriert und gegebenenfalls als redundanter Code markiert.

### 3.2.2 Besonderheiten von Kotlin

Neben den bereits bekannten Klassenarten: *class* und *abstract class*, erweitert Kotlin die Menge um *Object*[8] und *data class*[9]. *Data class* ist eine Klasse, deren Hauptaufgabe es ist Daten zu halten anstatt sie zu verarbeiten. Als Beispiel nehmen wir die Klasse *Node*, welche einen Knoten in unserer Projektstruktur darstellt.

Listing 3: Node

```
public data class Node
     private constructor(val members:MutableList<Node>,val label : String,
         val type : NodeType){
```

[8] https://kotlinlang.org/docs/reference/object-declarations.html
[9] https://kotlinlang.org/docs/reference/data-classes.html

```
    constructor(members:MutableList<Node>,label : String, type :
        NodeType, isSelectedForSharing : Boolean ):this(members, label,
        type){
       this.isSelectedForSharing = isSelectedForSharing
    }
    var isSelectedForSharing : Boolean? = null
    override fun toString() : String {
       return label + ''('' + type + '')''
    }
  ...
}
```

Die Klasse selbst wird mit dem Schlüsselwort *data class* eingeleitet. Wie bereits bei der Klasse *Contact* (siehe Listing 1), werden im Konstruktor die unveränderlichen Felder mit *val* definiert. Auffällig sind hier, die zwei Konstruktoren. Der private Primärekonstruktor ermöglicht uns, die Vorteile der *data class* zu nutzen. Was hier nicht zu sehen ist, ist die Implementierung von *equals()*, *toString()* und *hashCode()*. Dies ist bei einer *data class* auch nicht notwendig, denn diese werden vom Compiler automatisch implementiert. Dieser verwendet dazu die im primären Konstruktor angelegt Felder und implementiert diese Funktionen selbst. Aus diesem Grund legen wir hier den primären Konstruktor als *private* fest, sodass wir sicherstellen, dass *isSelectedForSharing* nie *null* annimmt und in den oben genannten drei Funktionen nicht verwendet wird. Da wir in unserem Beispiel mit der Funktion *toString()* lediglich das Label und den Typ des Knotens erhalten möchten, müssen wir diese überschreiben.
Eine weitere neue Klasse ist *Object*. Damit können in Kotlin Singletons bereitgestellt werden. Ein Beispiel aus der Reimplementierung ist die Klasse *JavaScriptAPI*.

Listing 4: die *JavaScriptAPI* als *object*

```
object JavaScriptAPI {
   private val LOG = Logger.getLogger(JavaScriptAPI::class.java)
   private val GSON = Gson()

   @JvmStatic
   fun showError(browser: IBrowser?, errorMessage: String?) {
      LOG!!.debug(
         (''Trigger js showError() on browser '' + browser!!.getUrl() +
             '',''
              + errorMessage)
      )
      triggerEvent(browser, ''showError'', errorMessage)
   }
   ...
}
```

Wie wir dem Listing 4 entnehmen können, wird diese Klasse nicht mit dem Schlüsselwort *class*, sondern mit *object* definiert. Auffällig in dieser Implementierung ist die Annotation *@JvmStatic*. Diese deklariert eine Methode oder Variable innerhalb der Klasse *object* als statisch. Falls wir nur einzelne Felder als statisch definieren wollen,

können wir das Schlüsselwort *const* verwenden. Für einzelne Funktionen biete Kotlin uns die Möglichkeit ein sogenanntes *companion object* zu erzeugen. Nehmen wir die Funktion aus dem Beispiel in Listing 5.

Listing 5: die *JavaScriptAPI* mit *companion object*

```
class JavaScriptAPI() {
   private val LOG = Logger.getLogger(JavaScriptAPI::class.java)

   companion object{
      @JvmStatic
      fun showError(browser: IBrowser?, errorMessage: String?) {
         LOG!!.debug(
            (''Trigger js showError() on browser '' + browser!!.getUrl() +
                '',''
                  + errorMessage)
         )
         triggerEvent(browser, ''showError'', errorMessage)
      }
   }
}
```

Ein *companion object* ist vergleichbar mit einer verschachtelten Klasse. Für das erstellte Objekt müssen wir jedoch keinen Namen vergeben. Der statische Aufruf einer Methode innerhalb eines solchen Objektes ist in Java und Kotlin identisch.
Kotlin bietet noch weitere Besonderheiten. In Listing 6 ein Beispiel in Java:

Listing 6: die Funktion *refine* in Java

```
public <T> T refine(Object argument, Class<T> targetType) {
    if (argument instanceof Boolean) {
      return getTypedArgument((Boolean) argument, targetType);
    }
    if (argument instanceof Double) {
      return getTypedArgument((Double) argument, targetType);
    }
    if (argument instanceof String) {
      return getTypedArgument((String) argument, targetType);
    }
    if (argument == null) {
      return null;
     ...
}
```

und im Vergleich dazu mit Kotlin in Listing 7.

Listing 7: die Funktion *refine* in Kotlin

```
@JvmStatic
public fun <T> refine ( argument : Any?, targetType: Class<T> ): T?{
   if(argument == null){
      return null
   }

   when(argument){
      is Boolean -> return getTypedArgument(argument, targetType)
      is Double -> return getTypedArgument(argument, targetType)
      is String -> return getTypedArgument(argument, targetType)
   }
   ...
}
```

Mit *when* können wir mehrfaches Aufrufen von *if... else* vermeiden.
Weitere Einsparungen bietet der sogenannte Elvis-Operator *?:*. Ist der Wert links vom Operator *null*, wird der rechte verwendet. Zum Beispiel *var* isAvailable = *Parameter ?: false*. Ist Parameter *null* wird *isAvailable* false zu gewiesen.
Um vor einem Methodenaufruf zu prüfen, ob das Objekt den Wert *null* angenommen hat, gibt es in Kotlin zwei Möglichkeiten. Die Erste ist ein sogenannter Safe Call. Dieser wird in Aufrufketten verwendetet, zum Beispiel *Vater?.Kind?.Name*. Mit dem *?*-Symbol stellen wir sicher, dass der Aufruf nur ausgeführt wird, wenn der Wert nicht *null* ist, ansonsten springt das Programm zur nächsten Befehlszeile. Soll stattdessen eine *NullPointerException* geworfen werden, so nutzen wir die zweite Möglichkeit den *!!*-Operator. Dieser wandelt den Typ für den Methodenaufruf in ein *NonNull*-Typ um.

### 3.2.3 Kotlin und Reflection

Ein für uns wichtiges Thema ist Reflection. Kotlin unterstützt dies auch für Java.

> **Reflection**
> Unter Reflection versteht man einen Verbund von Funktionen und Bibliotheken, die es ermöglichen die Struktur des Programms, während der Laufzeit, zu untersuchen.

Um eine Kotlin-Klasse als Objekt zu erhalten, schreiben wir Klassenname*::class*. Möchten wir eine Kotlin-Klasse als Java-Klasse zurückgeben, schreiben wir Klassenname*::class.java*. Als Beispiel können wir hier die Methode *createPages()* der Klasse *HTMLUIContextFactory* im Listing 8 betrachten.

Listing 8: Beispiel für Reflection: die Funktion *createPages()*

```
private fun createPages() {
   add(
      AccountPage::class.java, MainPage::class.java,
          SessionWizardPage::class.java,
      ConfigurationPage::class.java
   )
}
```

Mit *createPages()* rufen wir die Methode *add* auf. Diese fügt die Klassen dem PicoContainer hinzu. Hierfür benötigen wir jedoch Java-Klassen. Aus diesem Grund übergeben wir der Methode *add* mit *AccountPage::class.java*, die zu unser Kotlinklasse gehörende Java-Klasse.

Um Reflection verwenden zu können, müssen wir auch bei Kotlin eine separate Bibliothek einbinden, die *kotlin.reflect*. Stellenweise müssen wir beim Import jedoch die Klasse, die wir nutzen wollen, direkt angeben zum Beispiel *import kotlin.reflect.KType*. Da dies ausschließlich bei Eclipse notwendig ist, schließe ich auf einen Defekt im Plug-in.

### 3.2.4 Kotlin und Nebenläufigkeit

Kotlin kann im Bezug auf die Nebenläufigkeit ebenfalls die Bibliotheken von Java verwenden, nutzt jedoch einen eigenen Weg Nebenläufigkeit umzusetzen. Hierbei müssen wir jedoch einige Dinge beachten. Das Schlüsselwort *synchronized* gibt es in Kotlin ausschließlich nur im Zusammenhang mit einem Lock. Es kann jedoch auch als Annotation einer Methode verwendet werden und erfüllt dann denselben Zweck wie das Schlüsselwort in Java.
Alle Objekte in Kotlin erben von der Klasse Any. Diese ist der Klasse Object aus Java ähnlich. Es fehlen ihr jedoch die wichtigen Methoden: *notify()*, *notifyAll()* und *wait()*. Um diese Methoden verwenden zu können, wird ein separates Lock benötigt. Als Beispiel in Listing 9 nutzen wir einen Auszug aus der Klasse *BrowserManager*.

Listing 9: *notifyAll()* implementiert in Kotlin

```
private val lock = java.lang.Object()

fun setBrowser( page: IBrowserPage?,browser: IJQueryBrowser?) =
synchronized(lock){
      if(browser != null)
         browsers.put(page!!::class, browser)
         for (renderer in page!!.renderers!!) {
            renderer!!.addBrowser(browser)
         }
         lock.notifyAll()
}
```

In Java hätten wir in der Definition der Methode *synchronized* verwendet, welches *this* automatisch als Lock nutzt.
*Runnable*s sind in Kotlin und in Java Singletons. Während wir in Java ein *Runnable* mit *new Runnable()* erzeugen können, wird in Kotlin ein anonymes Objekt verwendet, welches von der Klasse *Runnable* erbt. Wie in Java müssen wir auch in Kotlin die *run()*-Methode überschreiben. Ein entsprechendes Beispiel können wir der Klasse *BrowserManager* entnehmen.

Listing 10: JavaScriptAPI

```
fun showDialogWindow(browserPage: IBrowserPage?) {
   val runnable = object : Runnable {
      override fun run() {
         ...
   }
   uiSynchronizer!!.asyncExec(runnable)
}
```

Die oben genannten Möglichkeiten zeigen uns, wie die aus Java bekannte Logik in Kotlin umgesetzt werden kann. Kotlin bietet noch andere Wege, mit Nebenläufigkeit umzugehen und verwendet hierfür den Begriff Koroutine. Dieser umfasst eine Sammlung an High-Level-Funktionen und Methoden, die uns die Arbeit mit Threads erleichtern sollen. Dies würde jedoch bedeuten, dass wir das Core-Module entsprechend in Kotlin schreiben müssten, da hier die Methoden implementiert sind, die Threads erstellen und verwalten.

### 3.2.5 Testen mit Kotlin

Um die Implementierung von Tests in Kotlin zu erleichtern, stellen uns die Entwickler die Bibliothek *kotlin.test* bereit. Diese bietet uns unterschiedliche Funktionalitäten. Für die Reimplementierung können wir in diesem Falle die bestehenden Unit-Tests nutzen. Lediglich kleinere Änderungen beim Aufrufen der Methoden der Klasse *TypeRefinery*, mussten angepasst werden. Alle anderen Tests können wir unverändert weiterhin verwenden. Mit dem aktuell herangezogenen Framework würde für uns eine Implementierung der Tests in Kotlin lediglich mehr Aufwand bedeuten.

## 3.3 Die Modellklassen

Nach der Einarbeitung in die Programmiersprache Kotlin, arbeitete ich mich tiefer gehend in den Quellcode des Saros Plug-ins ein. Dabei stieß ich auf folgende Klassen, die sowohl in Java als auch in JavaScript implementiert wurden:

- ProjectTree inklusive ProjectNode und
- State, die im Frontend als JSON-Objekt verarbeitet werden, sowie
- Contact und

- Account, die im Modul *UI.Frontend* unter *html/src/utils/propTypes.jsx* implementiert werden.

Einzig die Klasse *ProjectTree* ließ sich nicht störungsfrei implementieren. Kotlin erlaubte es mir nicht, die Klasse Enum in einer verschachtelten Klasse zu definieren. Somit musste ich die Klasse *Type*, aus der verschachtelten Klasse *Node* in der Oberklasse *ProjectTree* implementieren. Die weiteren Klassen ließen sich ohne nennenswerte Probleme in Kotlin neu implementieren.

## 3.4 Die Kommunikation zwischen HTML-GUI und Saros-Core

Die Kommunikation zwischen der HTML-GUI und dem Saros-Core wird durch zwei Adapter ermöglicht. Die *JavaScriptAPI* ist der Adapter, der das UI.Frontend-Modul über die Änderungen aus dem Core informiert. Hierzu werden die notwendigen Daten in Form von Objekten der Klassen *state*, *account* und *project* in den Datentyp JSON umgewandelt. Die Daten werden im JSON-Format in eine JavaScriptfuntkion eingefügt. Anschließend wird diese vom Browser ausgeführt.

Der zweite Adapter besteht aus mehreren Klassen. Zur Vereinfachung fasse ich diese Klassen unter dem Namen *browserfunctions* zusammen. Diese werden durch Invoking 3.1 ausgeführt. Die Methoden werden in JavaScript mit den entsprechenden Parametern aufgerufen und in Java ausgeführt.

Beide Adapter waren zu Beginn in Java implementiert. Ich untersuchte, ob Möglichkeiten existieren, die internen Abläufe für zukünftige Entwickler verbessern könnten. Dabei stellte ich jedoch fest, dass die aktuelle Methode ebenfalls in Kotlin verwendet werden muss. Aus diesem Grund habe ich die bestehende Logik ebenfalls in Kotlin implementiert. Ich empfand die ursprüngliche Bezeichnung der *browserfunctions* mit *__java_Methodenname* als irreführend und änderte diese zu *__kotlin_Methodenname*.

## 3.5 Integration in Gradle

Um Kotlin in den Build-Prozess von Gradle einzubinden, benötige ich einige Schritte. Zuerst binde ich mit *plugins id "org.jetbrains.kotlin.jvm" version "1.3.11"* das Plug-in ein, welches Kotlin in JVM-Bytecode kompiliert. Um die Modellklassen ebenfalls in JavaScript auszugeben, binde ich mit *apply plugin: "kotlin2js"* das notwendige Plug-in ein. Ohne weitere Angaben kompiliert dieses Plug-in alle Kotlindateien auch in JavaScript. Da ich jedoch lediglich die Modellklassen benötige, muss ich zusätzlich noch den Dateipfad angeben. Ein Beispieleintrag könnte so aussehen: *sourceSets  main.kotlin.srcDirs += 'src/main/myKotlin'*.

## 3.6 Probleme

Bei der Neuimplementierung stieß ich immer wieder auf neue Herausforderungen und kleinere Fallstricke, welche die neue Sprache mit sich brachte. Im kommenden Abschnitt gehe ich auf die größten von ihnen ein und wie ich sie lösen konnte.

### 3.6.1 Herausforderung "Debuggen"

Das Debuggen war eine der größten Herausforderungen. Da es sich bei Saros um ein Plug-in handelt, ließ sich die Debug-Funktion meist nicht verwenden. Mit dem Beginn des Debug-Modus startet eine neue Instanz von Eclipse. Auf diese zusätzliche Instanz konnte ich mit dem Debug-Modus nicht zugreifen. Nach einer Reimplementierung traten häufig Fehler auf. Die Fehlermeldung, die am häufigsten auftratt, ist die *"java.lang.ClassNotFoundException"*. Nach einigen Nachforschungen in den Errorlogs und der Verwendung von Tests, stieß ich jedoch auf die eigentliche Ursache dieser Exception. Denn in den meisten Fällen war diese Fehlermeldung irreführend. Häufig war eine *NullPointer-* oder eine *TypeCastException* die eigentliche Ursache. Ein kleines Beispiel: Die Klasse *AccountPage* erbt von *AbstractBrowserPage*. Im PicoContainer soll nun ein Objekt der Klasse *AccountPage* erstellt werden. Dies führt zum Aufruf des Konstruktors der Klasse *AbstractBrowserPage*. Wird innerhalb dieses Aufrufes eine TypeCast- oder NullPointerException geworfen, kann das Objekt *AccountPage* nicht erstellt werden. Der PicoContainer registriert nur das fehlende Elternobjekt und gibt anstelle der ursprünglichen *TypeCast-* oder *NullPointerException* eine *"java.lang.ClassNotFoundException"* zurück. Um das Problems zu lösen, ist es erforderlich das Tests der Konstruktoren implementiert werden, die den PicoContainer nicht verwenden.

### 3.6.2 Fehler mit dem SWT-Browser

Bereits beim ersten Starten der HTML-GUI stieß ich auf eine Herausforderung. Um die Logik hinter dem Frontend bearbeiten und testen zu können, musste ich zuerst das Frontend bauen. Dank der Anleitung meines Vorgängers ging dies schnell und unkompliziert. Doch bereits beim Start stellte ich fest, dass die grafische Benutzeroberfläche nicht dargestellt werden kann und folgende Fehlermeldung erschien: *"Saros couldn't initialisieren the SWT browser widget to display HTML UI"*. Der Fehler schien lediglich Ubuntu zu betreffen, da andere Mitglieder des Teams mit Windows arbeiteten und dort keine Fehler auftraten. Mehrere Lösungsansätze, wie unter anderem das Installieren vermeintlich fehlender Bibliotheken und das Setzen von Umgebungsvariablen im System, schlugen fehl. Letztendlich brachte lediglich ein Update auf Eclipse 4.4 die Lösung.

### 3.6.3 Defekte im Plug-in von Kotlin für Eclipse

Kotlin ist eine Programmiersprache, die von JetBrains entwickelt wurde. Wir erhalten die beste Unterstützung bei der Implementierung mit der Entwicklungsumgebung IntelliJ IDEA. Für die Entwicklung in Eclipse steht uns ein entsprechendes Plug-in bereit, welches stetig weiterentwickelt wird. Gerade zu Beginn meiner Entwicklung gab es Schwierigkeiten Kotlin-Klassen im PicoContainer erstellen zu lassen. Dieser akzeptiert ausschließlich Java-Klassen. Es ist aber möglich, Kotlin-Klassen als Java-Klassen weiter zu geben, siehe auch Kapitel 3.2.3, jedoch schien dies in diesem Zusammenhang nicht zu funktionieren. Nach dem ich mehreren Tage recherchierte und einige Lösung fehlschlugen, konnte ich das Problem mit einem Update des Plug-ins beheben.

Kurz darauf stieß ich auf einen weiteren Defekt, der gelegentlich auftritt. Nach einem Update müssen die entsprechenden Kotlin-Bibliotheken, in meinem Fall Kotlin Nature, entfernt und erneut zum Projekt hinzugefügt werden. Danach kompiliert alles wie gewohnt. Dieser Fehler kann ebenfalls auftreten, wenn Eclipse häufig neu gestartet wird.

Kein konkreter Defekt, jedoch ein Nachteil gegenüber IntelliJ IDEA, ist die fehlende Autovervollständigung.

Eine weitere Auffälligkeit bemerkte ich beim Thema Reflection. Mit Klassenname::*class.java* sollte Kotlin die äquivalente Java-Klasse übergeben. Bei der Klasse *Long* schlägt dies jedoch fehl. Dies bemerkte ich im Zusammenhang mit den JUnit-Tests der *TypeRefinery*. Da ich die *TypeRefinery* nach der Implementierung der Klasse *TypedJavascriptFunction* entsprechend auf Kotlin angepasst habe, löste sich das Problem damit ebenfalls. Die grundlegende Logik der *TypeRefinery* blieb erhalten, jedoch musste ich die Parametertypen von Java-Klassen auf Kotlin-Klassen umgeschrieben werden.

# 4 Fazit

In diesem Kapitel fasse ich meine Arbeitsergebnisse zusammen und anschließend werde ich diese rückblickend bewerten.

## 4.1 Zusammenfassen der Arbeitsergebnisse

Die Suche nach möglichen Technologien, um das Problem aus 1.3 zu lösen, ergab Fantom, Scala und Kotlin als mögliche Kandidaten. Diese boten die Möglichkeit die Anforderungen aus 2.1 zu erfüllen, jedoch mit unterschiedlich hohem Aufwand.

Bei der Untersuchung von Fantom stellte ich fest, dass dieser Kandidat aus mehreren Gründen ungeeignet ist. Als erstes gibt es für Fantom weder ein Plug-in für Eclipse, noch existiert ein aktuelles Plug-in für IntelliJ IDEA. Ein weiterer negativer Aspekt ist der zusätzliche Aufwand, Fantom in den Build-Prozess zu integrieren. Dieser ist im Vergleich zu den anderen Kandidaten um einiges höher. Des Weiteren war die Einarbeitung aufgrund der geringen Dokumentation schwer.

Scala erfüllte, in Kombination mit dem Framework Scala.js, ebenfalls die Anforderungen. Es ist kompatibel zu Java, lässt sich in den Build-Prozess integrieren und mit Scala.js kann Scala auch in JavaScript kompiliert werden. Lediglich bei der Einarbeitung stellte ich fest, dass die Komplexität die Scala mit sich bringt, den Einstieg für nachfolgende Entwickler nicht vereinfacht.

Kotlin ist kompatibel mit Java. Klassen und Methoden, die mit Kotlin implementiert werden, lassen sich ebenfalls in JavaScript kompilieren. Durch das bestehende Plug-in für Kotlin ist die Integration in den Build-Prozess von Ant und Gradle ohne große Aufwände möglich. Die Evaluation ergab für mich, dass Kotlin der geeignete Kandidat zur Implementierung ist.

Das UI-Modul konnte ich vollständig mit Kotlin implementieren. Während ich die

Modellklassen mit Kotlin zügig und ohne nennenswerte Aufwände programmieren konnte, fielen mir bei den komplexeren Klassen die Besonderheiten von Kotlin auf. Ein mehrfaches Auftreten von Funktionsrümpfen, war trotz näherer Untersuchung des Quellcodes für mich nicht ersichtlich.

Zur Qualitätssicherung konnte ich die bestehenden Unit-Tests verwenden. Hier musste ich lediglich kleinere Anpassungen vornehmen. Des Weiteren fließt die Dokumentation der Modellklassen und die Dokumentation des Quellcodes mit in die Sicherung der Qualität ein.

## 4.2 Rückblick und Auswertung der Arbeitsergebnisse

Rückblickend betrachtet war Kotlin die richtige Entscheidung. Mit dieser Arbeit konnte ich zeigen, dass Kotlin alle gestellten Anforderungen erfüllt. Die Einarbeitung in und Dokumentation von Kotlin zeigten auch, dass der Umstieg auf diese Programmiersprache keine große Hürde darstellt. Es zeigte sich auch, dass der Aufwand zur Integration in den Build-Prozess von Gradle kaum nennenswert ist.
Dafür war in meinen Augen der Aufwand bei den aufgetretenen Problemen größer als ich erwartet habe. Die Entwicklungsumgebung unter Ubuntu einzurichten hat mehr Zeit in Anspruch genommen, als ich vorab eingeplant hatte. Dieses Problem bleibt zukünftigen Entwicklern vorerst erspart, da bereits eine neuere Version von Eclipse und auch Java 1.8 verwendet werden kann. Die in meiner Arbeit festgehaltenen Auffälligkeiten des Kotlin-Plug-ins für Eclipse sind jedoch für kommende Entwickler von Bedeutung.

Die neue Lösung bringt auch einige Vorteile mit, die ich vorab nicht betrachtet habe. Kotlin ermöglichte mir den Quellcode an einigen Stellen zu verringern, ohne an Verständlichkeit einzubüßen. Die neue Möglichkeiten mit Nebenläufigkeit zu arbeiten, die Kotlin liefert, möchte ich an dieser Stelle ebenfalls positiv hervorheben. Hierbei handelt es sich um High-Level-Funktionen, sogenannte Koroutinen, mit denen Threads angenehmer erstellt und verwaltet werden können.

Im Nachhinein betrachtet ist der Aufwand für die derzeitigen vier Modellklassen verhältnismäßig hoch. Da das vollständige UI-Modul neu implementiert werden musste, wenn auch die Logik im Modul weitestgehend erhalten blieb. Trotz allem bin ich der Meinung, dass sich der Aufwand gelohnt hat, denn aktuell ist die HTML-GUI noch ein Prototyp. Es werden zukünftig noch weitere Funktionen benötigt, um die aktuelle GUI ersetzen zu können. Dies bedeutet, dass noch weitere Modellklassen folgen können. Damit habe ich ein Fundament für zukünftige Arbeiten an diesem Projekt geschaffen.

## 4.3 Zusammenarbeit mit dem Saros-Team

Für mich war die Zusammenarbeit im Saros-Team eine positive Erfahrung. Neben dem wöchentlichen Meeting mit dem Betreuer bietet die Arbeit im Entwicklerteam von Saros zwei Stand-Up-Meetings die Woche. In diesen wurde kurz der eigene aktu-

elle Stand beschrieben und welche Änderungen seit dem letzten Meeting stattfanden. Auch die zukünftigen Schritte wurden kurz vorgestellt und gegebenenfalls hilfreiche Denkanstöße gegeben. Auch konnte ich Fragen, die während der eigenen Arbeit aufkamen, im Team besprechen und es wurde gemeinsam versucht eine Lösung zu finden.

# 5 Ausblick

In diesem Kapitel gebe ich einen Ausblick darauf, wie die Ergebnisse meiner Arbeit weiterverwendet werden können. Abschließend möchte ich noch die ausstehenden Aufgaben nennen und erläutern.

## 5.1 Kotlin für das UI.Frontend-Modul

Um meine Arbeitsergebnisse optimal nutzen zu können, sehe ich es als unabdingbar an, dass das UI.Frontend-Modul ebenfalls in Kotlin implementiert werden sollte. Dann ließe sich die Struktur in UI-Modul, UI.Frontend-Modul und ein Modul SharedCode aufteilen. Hierbei können Klassen, die sowohl für die JVM als auch für JavaScript benötigt werden, in einem SharedCode-Modul untergebracht werden. Dies würde meiner Meinung nach die Einarbeitung in den Quellcode von Saros und die ablaufenden Prozesse erleichtern. Aktuell sind Unit-Tests für JavaScript und Java mit unterschiedlichen Frameworks implementiert. Dies wäre anschließend nicht mehr notwendig. Ein entsprechender Wrapper ermöglicht die Verwendung von Kotlin und React. Dieser wird derzeit eingesetzt und kann auch im Zusammenhang mit Kotlin weiterhin verwendet werden.

## 5.2 Umstieg auf Kotlin

Eine weitere Möglichkeit die Ergebnisse dieser Arbeit zu nutzen, ist Kotlin im gesamten Projekt zu verwenden. Ich denke Kotlin bringt das Potenzial mit, viel Code einzusparen und durch die zusätzlichen Methoden das Verständnis der Prozessabläufe innerhalb des Saros-Plug-ins zu verbessern. Auch lässt sich durch Kotlin die Nebenläufigkeit angenehmer verwenden, da Kotlin dafür High-Level-Funktionen bereitstellt. Diese Betrachtung könnte in einer separaten Arbeit durchgeführt werden. Der notwendige Aufwand das Plug-in vollständig in Kotlin zu implementieren ist jedoch immens.

## 5.3 Offene Arbeiten

Nach aktuellem Stand ist meine Implementierung des UI-Moduls noch im Zustand des Reviews. Hierzu legte ich zwei Branches im Repository von Saros an. Im *Development*-Branch befindet sich der aktuelle Fortschritt meiner Implementierung. Um den Mehrwert im Team diskutieren zu können, legte ich zusätzlich einen *Discussion*-Branch an. In diesem Branch haben das Saros-Team und ich die Möglichkeit

den Mehrwert und das weitere Vorgehen mit den Arbeitsergebnissen zu diskutieren. Die Reviews erfolgen durch die Pull Requests vom Branch *Development* in den *Discussion*-Branch. Bevor die Arbeit mit dem *Master*-Branch zusammengeführt wird. Ein weiterer offener Punkt ist die Integration in den Build-Prozess von Gradle. Dieser wird zwar konzeptionell in dieser Arbeit besprochen, konnte jedoch während dieser Zeit nicht umgesetzt werden. Der Grund dafür ist die im Verhältnis zur Arbeit, späte Einführung des Gradle-Build-Prozesses. Die Integration in diesem Build-Prozess möchte ich noch vor der Verteidigung umsetzen.

# Literaturverzeichnis

[1] Christian Cikryt. Evaluating the Use of a Web Browser to Unify GUI Development for IDE Plug-ins. Masterarbeit, Freie Universität Berlin, Inst. für Informatik, 2015.

[2] Martin Fowler. Inversion of Control Containers and the Dependency Injection pattern, January 2004.

[3] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. An overview of the scala programming language. Technical report, École Polytechnique Fédérale de Lausanne, 2004.

[4] Marius Schidlack. Neuimplementierung und Weiterentwicklung der HTML-GUI von Saros. Masterarbeit, Freie Universität Berlin, Inst. für Informatik, 2017.

[5] Bastian Sieker. User-Centered Development of a JavaScript and HTML-based GUI for Saros. Masterarbeit, Universität Paderborn, Machine Interaction and Software Technology Research Group, 2015.

[6] Raoul-Gabriel Urma. Alternative Languages for the JVM, July 2014.

[7] Nina Weber. Einstiegserleichterung für die Weiterentwicklung und Erweiterung der JavaScript- und HTML-GUI von Saros. Masterarbeit, Freie Universität Berlin, Inst. für Informatik, 2016.