Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Cornelius Buschka

Model-View-Presenter auf dem Desktop mit Embedded Browser und Java-Presenter

Cornelius Buschka

Model-View-Presenter auf dem Desktop mit Embedded Browser und Java-Presenter

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang Angewandte Informatik
am Studiendepartment Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuende Prüfererin : Prof. Dr. Birgit Wendholt
Zweitgutachter : Prof. Dr. Gunter Klemke

Abgegeben am 13.8.2012

Cornelius Buschka

Thema der Bachelorarbeit

Model-View-Presenter auf dem Desktop mit Embedded Browser und Java-Presenter

Stichworte

Model View Presenter (MVP), Grafische Benutzeroberfläche, Document Object Model (DOM), HTML, CSS, Desktop-Anwendung

Kurzzusammenfassung

HTML, CSS und die modernen Browser werden immer leistungsfähiger. Um hiervon zu profitieren wird ein Rahmenwerk entwickelt, das die Erstellung von grafischen Benutzer-Oberflächen in Java-Desktop-Anwendungen mit eingebettetem Browser ermöglicht. Hierzu wird das DOM (Document Object Model) des eingebetteten Browsers der Java-Anwendung über eine DOM-Bridge zur Verfügung gestellt. Zur Evaluation des Rahmenwerks wird ein Prototyp erstellt. Dabei wird das Model-View-Presenter-Muster angewandt, um die Unabhängigkeit der in Java implementierten Präsentationslogik von der Oberflächen-Technologie zu gewährleisten.

Cornelius Buschka

Title of the paper

Model-View-Presenter in Desktop Applications with Embedded Browser and Java-Presenter

Keywords

Model View Presenter (MVP), Graphical User Interface, Document Object Model (DOM), HTML, CSS, Desktop-Application

Abstract

HTML, CSS and modern browsers become more and more powerful. To profit from this a framework is developed which allows for the creation of graphical user interfaces in Java desktop applications with an embedded browser. For this the document object model (DOM) of the embedded browser is made available to the Java desktop application through a DOM-bridge. For the evaluation of the framework a prototype is built which uses the Model-View-Presenter-Pattern to ensure the independency of the presentation logic implemented in Java from interface technology.

Inhaltsverzeichnis

Abbildungsverzeichnis

1. Einführung

1.1. Motivation

Grafische Benutzeroberflächen (GUIs) von Software-Systemen werden immer aufwändiger gestaltet. Zu den üblichen Eingabeelementen wie z.B. Textfeldern und Schaltflächen kommen neue selbst entwickelte Bedienelemente, 3D-Effekte oder Animationen hinzu, um dem Benutzer die Bedienung einfacher und attraktiver zu machen. So setzen moderne Betriebssysteme bei ihren Oberflächen, z.B. OS X Aqua, Windows Aero oder Ubuntus Unity, intensiv visuelle Effekte ein, um die Aufmerksamkeit des Benutzers zu erregen oder zu signalisieren, mit welchen der dargestellten Elemente er interagieren kann. Auch Konsumenten-orientierte Webanwendungen versuchen über Werbe-Banner, grafische Hervorhebungen und über personalisierte Inhalte den Benutzer gezielt zu beeinflussen. Die Oberflächen auf mobilen Endgeräten werden für spezielle Interaktionsformen optimiert. Denn dort stehen manche Eingabemöglichkeiten nicht zur Verfügung (z.B. Mauseingabe) und müssen durch Alternativen (z.B. Gesten auf Touchscreens) ersetzt werden. Eine zeitgemäße Oberfläche ist also visuell aufwändig und passt sich an den jeweiligen Benutzer an.

Die Oberfläche besitzt für die Akzeptanz eines Software-Systems einen hohen Stellenwert. Der Endnutzer nimmt das Software-System weitgehend über die Benutzeroberfläche wahr. Daher wurd die wahrgenommene Qualität des Software-Systems maßgeblich durch die Qualität der Oberfläche bestimmt. Funktioniert die Oberfläche nicht, nimmt der Benutzer dies vom gesamten Software-System an. Entspricht eine Benutzeroberfläche nicht dem aktuellen Stand hinsichtlich der Gestaltungselemente und Interaktionsformen, wird der Nutzer dies auch vom für ihn nicht sichtbaren Teil des Systems annehmen.

Speziell Web-Oberflächen haben im letzten Jahrzehnt erhebliche Fortschritte gemacht. Die rasante Entwicklung wird durch die in Funktionsumfang und Verarbeitungsgeschwindigkeit immer leistungsfähiger werdenden Browser vorangetrieben. Das macht die Web-Technologien auch für die Verwendung auf dem Desktop interessant. Allerdings sind Web-Anwendungen ggü. Desktop-Anwendungen im Nachteil was die Integration mit lokalen Ressourcen, z.B. andere Programme auf dem lokalen Desktop-PC, angeht. Aufgrund des Browser-Sicherheitsmodells können Web-Anwendungen nicht (bzw. nur sehr eingeschränkt) darauf zugreifen. Hier gibt es verschiedene Lösungsansätze, wie u.a. native Plugins. Diese

stellen allerdings für die meisten Benutzer eine Hürde dar, da sie explizit installiert werden müssen und für IT-Abteilungen die automatisierte Softwareverteilung verkomplizieren. Desktop-Anwendungen und Web-Anwendungen haben also jeweils ihre Vor- und Nachteile.

Motivation dieser Arbeit ist die Modernisierung der Desktop-Oberfläche eines zehn Jahre alten sich in Produktionsbetrieb befindenden Softwaresystems aus der Versicherungsbranche. Über die Desktop-Anwendung interagieren Sachbearbeiter mit dem Bestandsverwaltungssystem. Die Geschäftslogik und Präsentationslogik des Systems sind in Java realisiert. Die eingesetzte Oberflächen-Technologie der Desktop-Anwendung ist das Standard Widget Toolkit (SWT, s. swt). Die Anwendung ist für den Desktop-PC konzipiert und mit weiteren Desktop-Anwendungen, u.a. Microsoft Word für die Bearbeitung des Schriftverkehrs mit Kunden, integriert. Um die Benutzerakzeptanz der Software zu erhalten, soll die Oberfläche modernisiert werden.

Eine Neuentwicklung der Anwendungsoberfläche als reine Web-Anwendung kommt aufgrund der engen Integration mit Desktop-Anwendungen und aus wirtschaftlichen Gründen nicht in Frage. Um von der schnellen Entwicklung im Webbereich und vom weit verbreiteten Know-How profitieren zu können, sollen die Web-Oberflächen-Technologien zur Modernisierung und weiteren Entwicklung der Desktop-Oberfläche eingesetzt werden.

Während in einer klassischen Web-Anwendung im HTTP-Request-Response-Modell gearbeitet wird, operieren moderne Ajax-Web-Anwendungen (vgl. Wikipedia AJAX 2012) aus JavaScript heraus direkt auf dem DOM des Browsers. Würde der Browser nun in eine Java-Desktop-Anwendung eingebettet und stünde das Browser-DOM in Java zur Verfügung, kann die Darstellung der Oberfläche auf der Basis von HTML und CSS erfolgen und die Präsentationslogik weiterhin in Java realisiert werden.

Die Nutzung der Web-Oberflächen-Technologien HTML und CSS zur Gestaltung der Oberfläche in einer Desktop-Anwendung, die mit lokalen Programmen integrierbar ist, unter Nutzung des ereignisorientierten Browser-DOMs aus Java heraus, stellt eine Lösung dar, die die Vorteile beider Welten vereint und eine schrittweise Migration der Oberfläche des vorliegenden Softwaresystems möglich macht.

1.2. Zielsetzung

Ziel dieser Arbeit ist es zu zeigen, ob und wie die Entwicklung von Desktop-Oberflächen mit HTML, CSS und Javascript als Oberflächen-Technologie durchgeführt werden kann. Dies soll am Beispiel eines vorliegenden Softwaresystems untersucht und dabei ein Migrationspfad für die schrittweise Modernisierung der Oberfläche berücksichtigt werden.

Die technischen Herausforderungen hierbei sind folgende:

- Für die Implementierung neuer Oberflächenteile unter Einsatz der Web-Oberflächen-Technologien HTML, CSS und JavaScript muss ein Web-Browser in die Desktop-Anwendung eingebettet werden. Dies soll so erfolgen, dass der Benutzer weiterhin den Eindruck hat, eine Desktop-Anwendung zu nutzen.
- Der Browser muss mit den notwendigen Ressourcen versorgt werden können, die er zur Darstellung der Oberfläche benötigt, wobei die Datenquelle hierfür auch ein externes System, wie z.B. ein Content-Management-System sein kann. Die Datenquelle muss also durch den Anwendungsentwickler bestimmt werden können.
- Um eine Migration zu ermöglichen, soll die in Java vorhandene Präsentationslogik weiterverwendet werden können. Die SWT-basierte Vorgehensweise und die neue auf eingebettetem Browser basierende müssen miteinander zusammenarbeiten können, um eine schrittweise Migration zu erlauben. Z.B. kann ein Oberflächen-Teil, der noch in der SWT-basierten Vorgehensweise realisiert ist, eine Liste aus Elementen darstellen. Ihr Inhalt kann wiederum über einen Editor, der auf der Browser-Vorgehensweise basiert, bearbeitbar sein.
- Die Lauffähigkeit des Softwaresystems auf den Plattformen Windows und Linux muss erhalten bleiben.

Ausgeschlossen werden Fragestellungen hinsichtlich der Sicherheit, die aus dem Einsatz eines Web-Browsers resultieren können. Da die geladenen Ressourcen vollständig lokal aus der Anwendung stammen und somit einen hohen Grad an Vertrauenswürdigkeit genießen, ist dieses Themengebiet in diesem Fall vernachlässigbar.

1.3. Vorgehensweise

Als erstes wird in Kapitel 2 der Stand der Entwicklung anhand verwandter Arbeiten und Technologien aufgenommen und geprüft inwiefern die Nutzung von Web-Oberflächen-Technologien für die Entwicklung von Desktop-Oberflächen bereits durchgeführt worden ist.

In der Analyse (Kapitel 3) wird eine Architekturvorlage für die Trennung der Präsentationslogik von der Oberflächen-Technologie erarbeitet, da dies den Austausch der Oberflächen-Technologie unter Erhalt der Präsentationslogik ermöglicht. Im Anschluss werden die funktionalen Anforderungen an ein Rahmenwerk zur Unterstützung der Entwicklung der Oberflächen von Desktop-Anwendungen auf Basis der Web-Oberflächen-Technologien anhand eines Anwendungsbeispiels abgeleitet.

Im Kapitel Design (4) wird ein Entwurf für das Rahmenwerk und einen Prototypen zur Evaluation der Anwendbarkeit des Rahmenwerks unter Nutzung der in der Analyse beschriebenen Architekturvorlage geliefert.

Das Kapitel Realisierung (5) beschreibt die Umsetzung von Rahmenwerk und Prototyp. Hier wird der Entwurf des Rahmenwerks konkretisiert und die Umsetzung anhand von Code-Ausschnitten beschrieben. Code-Ausschnitte des Prototyps belegen die Trennung der Präsentationslogik von der Oberflächen-Technologie.

Zum Schluss (Kapitel 6) werden alle Teilergebnisse zusammengefasst und es wird ein Fazit gezogen, in wie weit die gestellten Anforderungen erfüllt wurden und der erwartete Nutzen erreicht worden ist. Ferner werden Aspekte aufgeführt, die im Rahmen weiter führender Arbeiten behandelt werden könnten.

2. Verwandte Arbeiten

Im folgenden werden verschiedene Arbeiten darauf untersucht, ob sie sich mit der Erstellung von Desktop-Oberflächen mit den Web-Oberflächen-Technologien HTML und CSS beschäftigen oder ob sie dafür genutzt werden können. Als Resultat ergeben sich zwei Gruppen:

Die erste Gruppe nutzt den Browser als alleinige Laufzeitumgebung. Hierbei werden die technologischen Freiheiten, z.B. die Auswahl der Programmiersprache oder die Fähigkeit zur Integration mit anderen Programmen, so weit eingeschränkt, dass ein vernünftiger Migrationspfad nicht realisiert werden kann.

Die zweite Gruppe an Arbeiten verwendet den Browser als Komponente oder stellt ihn als solche zur Verfügung, so dass die Web-Oberflächen-Technologien für die Desktop-Oberfläche nutzbar gemacht werden können, ohne die technologischen Freiheiten einzuschränken.

Es stellt sich heraus, dass SWT ein Browser-Widget zur Einbettung eines Browsers und damit die erforderliche Basisfunktionalität bereitstellt, um die Web-Oberflächen-Technologien HTML und CSS in Kombination mit der Oberfläche einer Desktop-Anwendung nutzen zu können.

2.1. Arbeiten

2.1.1. JavaFX

JavaFX ist eine auf Java basierende Technologie für die Erstellung plattformübergreifender Anwendungen. Hierbei setzt Oracle auf der Verfügbarkeit der Java-Laufzeit-Umgebung auf zahlreichen Plattformen auf. JavaFX-Anwendungen laufen entweder als Java-Applet in der Applet-Sandbox oder werden über Java-WebStart als eigenständige Java-Applikation ohne Sandbox gestartet. Zur Entwicklung steht die Java-Plattform in vollem Umfang, bzw. durch die Einschränkungen der Sandbox begrenzt, zur Verfügung. Die Oberflächen werden entweder programmatisch in Form einer Scene (s. Javadoc javafx.scene.Scene) erzeugt oder über eine JavaFX-spezifische, auf XML basierende, DSL[1] beschrieben.

[1] Domain-Specific-Language, eine für einen speziellen Einsatzzweck entworfene Sprache

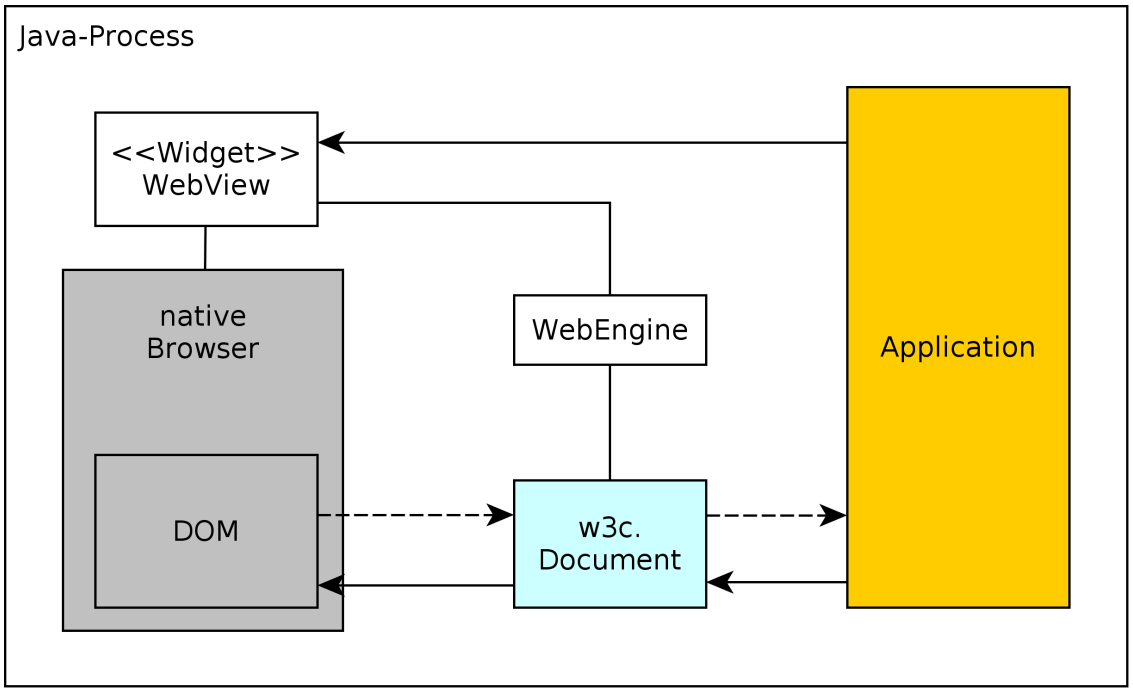


Abbildung 2.1.: JavaFX

JavaFX stellt mit der WebView (s. Javadoc javafx.scene.web.WebView) eine von der Plattform unabhängige Möglichkeit bereit, einen Browser einzubetten (s. 2.1). Beim eingebetteten Browser handelt es sich um den mitgelieferten Browser WebKit. Über die WebEngine (s. Javadoc javafx.scene.web.WebEngine) wird das Browser-DOM als Java-DOM-Document, das die W3C-DOM-API implementiert, zugänglich gemacht. Darüber sind sondierende wie auch verändernde Zugriffe möglich. Ebenfalls lassen sich Java-DOM-Event-Listener anmelden, die dann über die entsprechenden Ereignisse informiert werden.

JavaFX lässt die Strukturierung der Präsentationsschicht offen. So steht es einem frei, die Implementierung der Präsentationslogik in Java durchzuführen. Der Programmierer hat über das Java-DOM-Document Zugriff auf das Browser-DOM, das er so sondieren, Änderungen daran vornehmen und sich für Ereignisse registrieren kann.

Hiermit liefert JavaFX den Browser als Komponente, die sich sehr einfach in der vorhandenen Laufzeitumgebung einbinden lässt. Die Einbettung in die Oberflächen-Technologie Java Swing (oder seinem Vorgänger AWT) ist vorgesehen. Für die in der vorliegenden Anwendung genutzte Oberflächen-Technologie SWT ist ebenfalls eine Einbettung realisiert. Erste Implementierungen des Prototypen zeigten jedoch Darstellungsfehler, z.B. nicht scroll-bare Inhalte, obwohl Scrollbacken angezeigt waren, die den Einsatz der Technologie zum Zeitpunkt der Evaluation nicht zuließen.

2.1.2. Windows Presentation Foundation (WPF)

Die Windows Presentation Foundation ist Teil der .Net-Plattform von Microsoft. Die Verfügbarkeit der .Net-Plattform ist praktisch auf die Windows-Plattform begrenzt.[2] Die Oberflächen werden in einer auf XML basierenden DSL XAML beschrieben und per WPF dargestellt. Die Entwicklung erfolgt in einer der .Net-Sprachen, z.B. C#. WPF erlaubt auch die Einbettung eines Browser in Form des `WebBrowser-Widgets` (MSDN WebBrowser Class), um Web-Anwendungen anzuzeigen und auszuführen. Das DOM-Document des Browsers kann über die Objekt-Property `Document` des `WebBrowser`-Widgets erreicht werden. Das `Document` ist eine Browser-spezifische Implementierung des DOM, die über die Schnittstellen-Technologie COM exportiert wird.

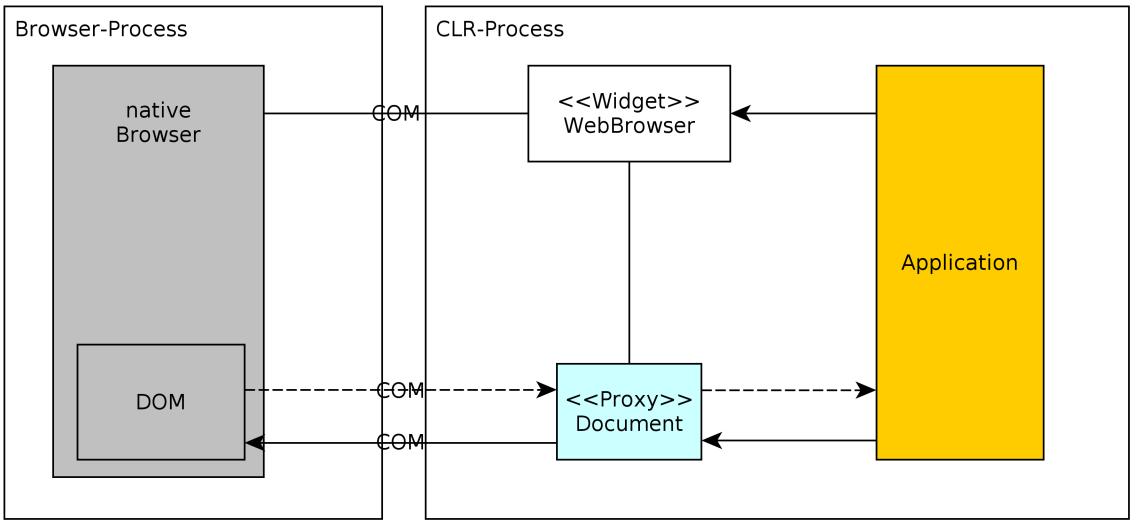


Abbildung 2.2.: Windows Presentation Foundation

Eine Realisierung der Oberfläche einer Desktop-Anwendung auf Basis des `WebBrowser`-Widgets würde analog zu der in JavaFX verlaufen: Die Anwendung erzeugt eine Oberfläche mit `WebBrowser`-Widget und sondiert und verändert den Browser-Inhalt über das DOM-Dokument. Auch Event-Registrierungen sind hier möglich. Lediglich die Browser-spezifische Schnittstelle des gelieferten `Document` ist möglicherweise problematisch. Genutzt wird hier der vom Betriebssystem mitinstallierte Browser Internet Explorer, der auch in seinen Versionen variieren kann. D.h. in der Entwicklung müssten verschiedene Browser-Versionen berücksichtigt werden.

[2] Es gibt zwar das Projekt mono (mono), das die .Net-Plattform auch unter Unix-ähnlichen Betriebssystemen bereitstellen will, allerdings ist die Kompatibilität zum Microsoft .Net nicht langfristig geklärt und es stehen auch bis heute nicht alle Funktionalitäten der Microsoft-.Net-Plattform unter mono zur Verfügung.

.Net stellt wie JavaFX den Browser als Komponente bereit. Wie die Arbeit damit geschieht ist dem Entwickler überlassen. D.h. es finden sich hier die selben Freiheitsgrade wie auch beim Einsatz von JavaFX. Die Notwendigkeit zur Berücksichtigung verschiedener Browser-Versionen ist aus Sicht des Entwicklers allerdings ungünstig und bedeutet Mehraufwand in der Entwicklung.

Die Nutzung von .Net für die vorliegende Anwendung würde eine Portierung oder eine Integration von .Net und Java notwendig machen. Die daraus resultierende Heterogenität an eingesetzten Technologien erschwert möglicherweise die Wartbarkeit des Softwaresystems.

2.1.3. Adobe Flex/ Air

Adobe Flex basiert auf der Flash-Player-Technologie. Der Flash-Player wird u.a. als Browser-Plugin verteilt und ist damit praktisch auf allen Desktop-Betriebssystemen und in allen verbreiteten Browsern vorhanden. Flex ist eine Erweiterung des Flash-Players um Bibliotheken und ein Oberflächen-Toolkit, das Komponenten zur Erstellung von Oberflächen mitliefert. Hierfür wurde die DSL MXML zur Beschreibung grafischer Oberflächen aus den von Flex mitgelieferten Bausteinen entwickelt. In ActionScript kann nun Präsentations- und Programmlogik implementiert werden. Da der Flash-Player im Browser ausgeführt wird, können Flex-Anwendungen sehr gut mit anderen Web-Anwendungen integriert werden. Es stehen auch Funktionen zum Austausch von Daten mit der umgebenden Webseite zur Verfügung (vgl. Adobe Livedocs ProgrammingHTMLAndJS).

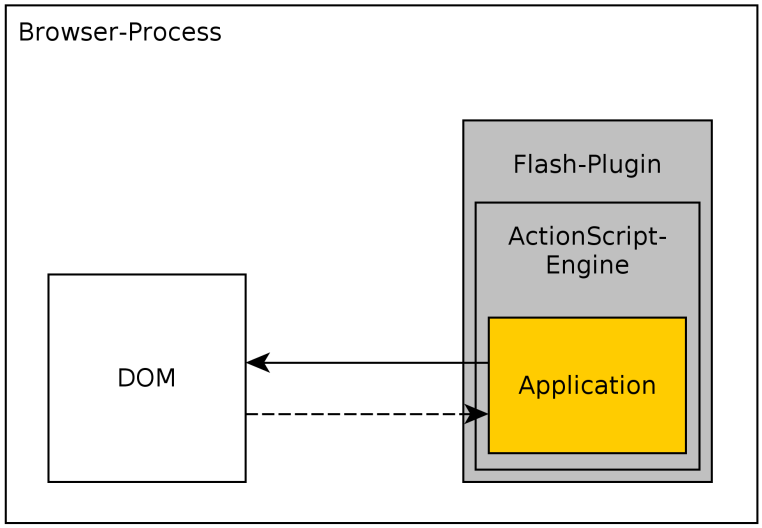
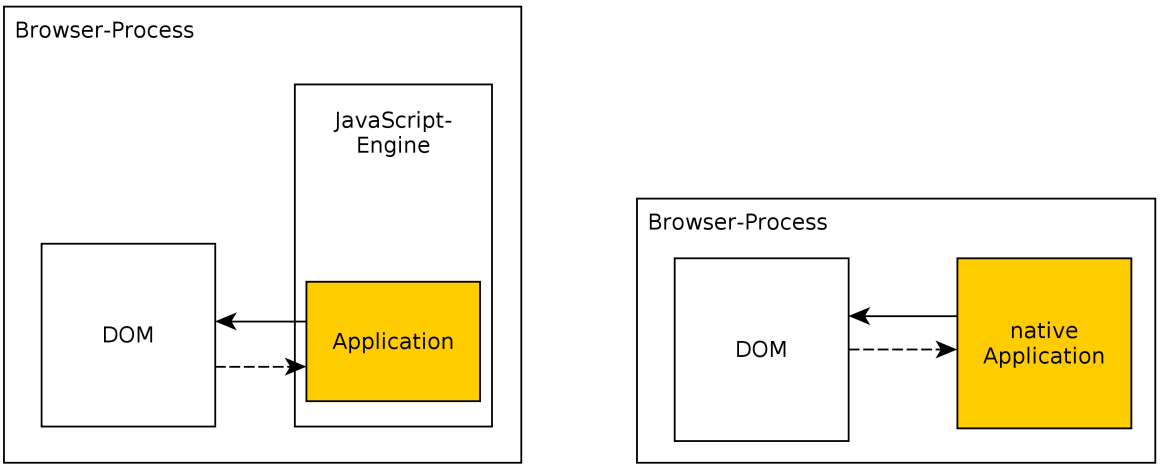


Abbildung 2.3.: Flash-Plugin als Gast im Browser

Der Flash-Player ist ursprünglich ein Interpreter zur Darstellung auf Vektorgrafik basierender Sequenzen, in Flash-Terminologie 'Movies' genannt. Um eine höhere Flexibilität zu erreichen, wurde der Player um ActionScript ergänzt, welches genau wie JavaScript auf dem EcmaScript-Standard basiert. Die vom Flash-Player selbst dargestellten Eingabeelemente fallen oft dadurch auf, dass sie sich visuell und vom Verhalten her von der Plattform unterscheiden. Z.B. sind Text-Inhalte in Flash-Movies oft nicht markier- und kopierbar oder das Scrolling-Verhalten unterscheidet sich von dem der restlichen Plattform. Dies wird vom Nutzer oft als störend wahrgenommen. Adobe Air dehnt Flex auf den Einsatz auf den Desktop aus, indem es plattformspezifische Funktionalitäten anbietet, die z.B. aus dem Browser heraus von Flex-Anwendungen aufgrund von Sicherheitsaspekten nicht erreichbar sind.

Eine Realisierung der grafischen Benutzeroberfläche einer Desktop-Anwendung auf Basis von Flex oder Air würde den Flash-Player als Plattform (entweder innerhalb des Browsers als Plugin oder außerhalb als AIR-Runtime) voraussetzen. Da keine anderen Programmiersprachen als ActionScript für die Plattform vorgesehen sind, müsste die Programmierung darin erfolgen.

2.1.4. XML User Interface Language (XUL)



(a) XUL mit Anwendung in JavaScript

(b) XUL mit nativer Anwendung

XUL (XUL), die XML User Interface Language, wurde vom Mozilla-Projekt entwickelt und findet auch u.a. im Mozilla Firefox z.B. für die Entwicklung von Add-Ons Anwendung. XUL definiert eine auf XML basierende DSL für die Beschreibung von Oberflächen. CSS wird zur Gestaltung genutzt und JavaScript zur Entwicklung der Präsentationslogik. XUL konnte sich seit 2001 nicht signifikant über die Grenzen der Entwicklung von Mozilla-Addons hinaus durchsetzen, obgleich mehrere Versuche, z.B. Luxor (Luxor), unternommen worden sind.

Dass sich XUL für die Realisierung von Desktop-Anwendungen eignet, zeigt z.B. der Music- und Video-Player Miro[3].

Ähnlich wie bei Flex würde die Entwicklung der Präsentationslogik in JavaScript stattfinden (alternativ auch in C oder C++) und die existierende Geschäftslogik evt. über eine Kommunikations-Schnittstelle wie XPCOM (vgl. xpcom) angebunden werden. Ein solche Integrationsszenario würde aufgrund der resultierenden Heterogenität an eingesetzten Technologien möglicherweise die Wartbarkeit des Softwaresystems erschweren.

2.1.5. Standard Widget Toolkit

Eclipse Standard Widget Toolkit (SWT, s. swt) ist ein auf mehreren Plattformen verfügbares Oberflächen-Toolkit für Java. SWT unterstützt die Einbettung eines Browsers über das Browser-Widget. Die Arbeitsweise mit dem eingebetteten Browser ist nicht vorgegeben. Das Browser-DOM wird nicht direkt zur Verfügung gestellt. Allerdings ermöglicht das Browser-Widget das Ausführen von JavaScript-Code in der JavaScript-Engine des Browsers und ermöglicht so die Sondierung und Veränderung des Browser-DOMs. Ebenfalls können Java-Callback-Objekte in den Objektraum der JavaScript-Engine so eingeblendet werden, dass Aufrufe aus der JavaScript-Engine heraus auf diese Java-Objekte möglich ist. So wäre z.B. die Registrierung von Event-Listener am Browser-DOM möglich.

Als eingebetteter Browser stehen Mozilla und WebKit (webkit) zur Auswahl. WebKit ist vollständig als Bibliothek enthalten und erfordert im Gegensatz zu Mozilla keine separate Installation. Damit steht derselbe Browser auf jeder Plattform in derselben Version zur Verfügung. Dies erleichtert die Entwicklung der Oberfläche in sofern, dass nicht verschiedene Browsertypen oder -versionen berücksichtigt werden müssen.

Da SWT die genutzte Oberflächen-Technologie in der vorliegenden Anwendung ist, gestaltet sich die Einbindung des SWT-Browser-Widget als einfach.

2.1.6. PhoneGap

PhoneGap ist ein Framework, um Anwendungen für mobile Endgeräte plattformübergreifend erstellen zu können, u.a. für Android, IOS. Hierbei wird eine Webseite mit Web-Technologien HTML, CSS und JavaScript erstellt und Plattfom-spezifisch paketiert. PhoneGap bindet ein Plattform-spezifisches Startmodul ein, das bei Anwendungsaufruf einen lokalen Browser startet und die Einstiegsseite lokal lädt. Um PhoneGap-Anwendungen die gleichen Möglichkeiten wie nativen Anwendungen zu geben, wird der Zugriff auf die APIs der Plattform

[3] http://getmiro.com/

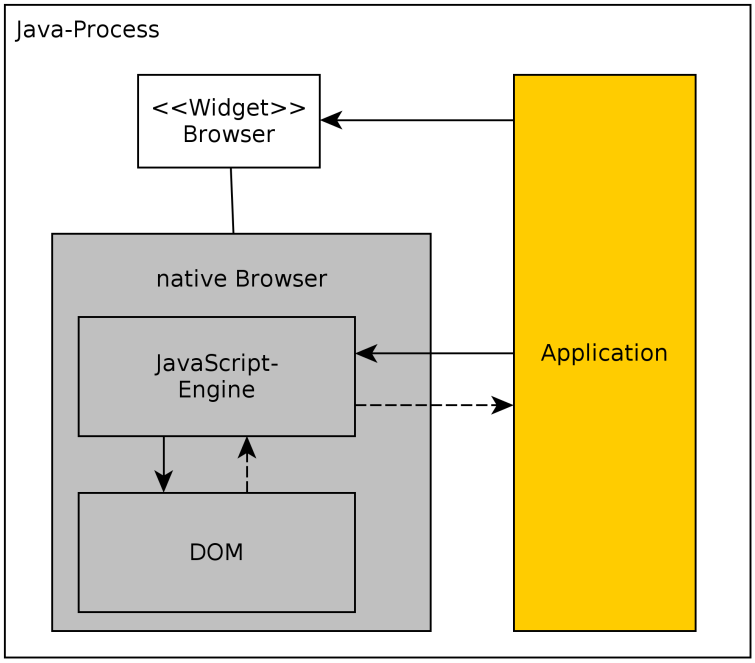


Abbildung 2.4.: Standard Widget Toolkit

über eine PhoneGap eigene API in JavaScript zur Verfügung gestellt. Hiermit erreicht PhoneGap einen hohen Grad an Plattformunabhängigkeit, obgleich eine Integration in die Umgebung möglich ist. Zwar stellen nicht alle Plattformen den gleichen Funktionsumfang zur Verfügung, dies stellt sich aber für den Programmierer wie ein Unterschied in der Gerätekonfiguration dar, die er sowieso berücksichtigen muss. Auf jeder Plattform sind also dieselben Technologien einsetzbar. Da es sich hier um die Standard-Web-Technologien HTML, CSS, JavaScript handelt, ist die Einstiegshürde sehr niedrig. Die Grenzen von PhoneGap für große Anwendungen hinsichtlich Wartbarkeit und Performance sind allerdings in der zwingenden Verwendung von JavaScript gesetzt.

2.1.7. ChameRIA

Bardin u.a. sehen die Vorteile zur Verwendung von Web-Technologien für die Entwicklung von Desktop-Anwendungen (vgl. Bardin u. a. 2011). Mit der ChameRIA wird eine komplette Web-Anwendung inklusive eines leichtgewichtigen Web-Servers und einem Browser in eine Anwendung eingebettet, um den Eindruck zu erwecken, dass es sich um eine einzelne Desktop-Anwendung handele. Die Oberfläche der Web-Anwendung ist mit HTML, CSS und JavaScript realisiert. Die Geschäftslogik ist in Form von SOA-Services bereits vorhanden. So

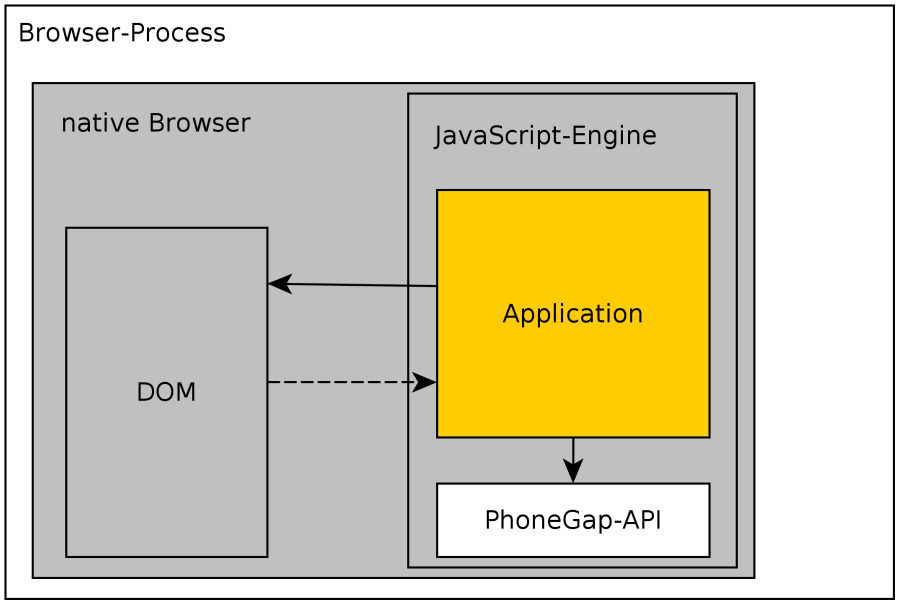


Abbildung 2.5.: PhoneGap

entwickeln Barding und Lalanda ein Protokoll, um über JavaScript die verfügbaren Services ansprechen zu können.

Das Ziel, HTML und CSS für die Entwicklung von Desktop-Anwendungen zu nutzen, wird damit erreicht. Bei ChameRIA wird allerdings die natürliche Schnittstelle zwischen Client und Server, die aufgrund der Kommunikation über Web-Sockets zur Geschäftslogik in Form von Services vorliegt, genutzt, um den Übergang von Web-Oberflächen-Technologien zur Java-Welt abzubilden (vgl. Diagramm 2.7(a)). Hieraus ergibt sich, dass die Präsentationslogik nur in JavaScript entwickelt werden kann, da keine andere Technologie (ohne weiteres) im Browser verfügbar ist. Aus Sicht des alternativen Deployment-Szenarios als Web-Anwendung ist dies auch sinnvoll. Eine solche Einschränkung gilt allerdings nicht für die Entwicklung am vorliegenden Softwaresystem. Wie in 2.7(b) gezeigt, soll die Präsentationslogik in Java vorliegen, d.h. der Schnitt zwischen Browser und Java wird zwischen DOM und Präsentationslogik und nicht zwischen Präsentationslogik und Geschäftslogik gezogen. Dies macht den von ChameRIA verfolgten Ansatz interessant, aber für den vorliegenden Zweck nicht brauchbar.

2.2. Bewertung

In folgender Tabelle werden die Eigenschaften der verschiedenen Arbeiten noch einmal zusammengefasst und bewertet:

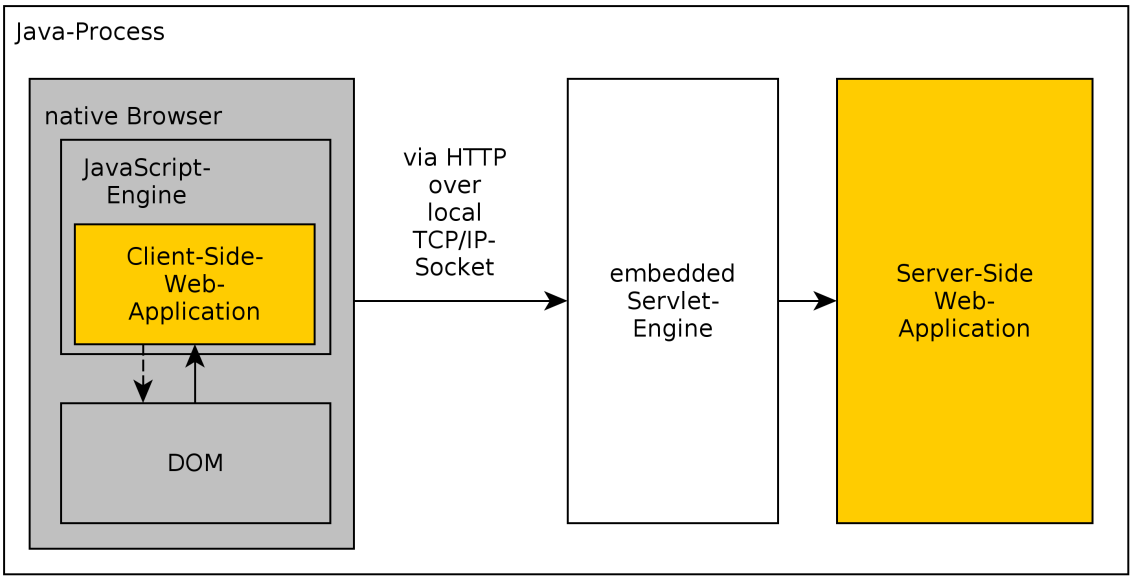
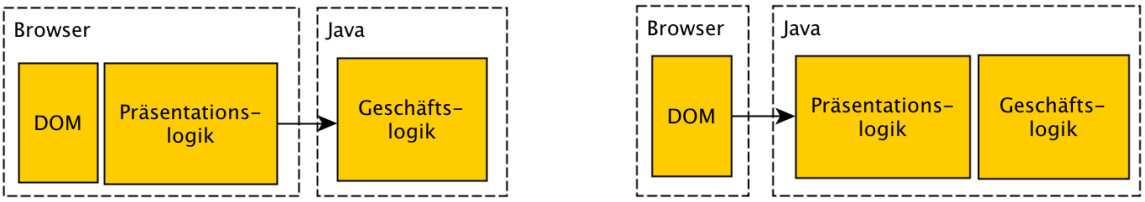


Abbildung 2.6.: ChameRIA



(a) bei ChameRIA (b) wie er für Desktop-Anwendung sein soll

Abbildung 2.7.: ChameRIA - Übergang vom Browser zu Java

Technologie	Plattform-verfügbarkeit	Integration mit Plattform-anwendungen	Entwicklungs-modell für Anwen-dung	Einschätzung der Zukunftsfähigkeit	erforderliche Maß-nahmen
JavaFX	Windows, Linux, Mac	möglich	Java, wie bisher	unsicher aufgrund niedriger Akzep-tanz von 1.x	warten auf Be-seitigung der Kinderkrankheiten, Bereitstellung von Resourcen
WPF	nur Windows	möglich	Entwicklung in .Net-Sprache	sicher	Portierung auf .Net
eclipse SWT	Windows, Linux, Mac	möglich	Java, wie bisher	sicher	Bereitstellung Browser-DOM und Resourcen
Adobe Flex/ Air	Windows, Linux, Mac	über Browser-Plugins	Impl. Oberfläche in JavaScript	unsicher	Portierung auf Ja-vaScirpt
Mozilla XUL	Windows, Linux, Mac	möglich	Impl. in C oder Ja-vaScript	unsicher	Portierung nach C oder JavaScript
PhoneGap	Andoid, iOS	möglich	Entwicklung in Ja-vaScript	unsicher	Portierung Phone-Gap auf Desktop und Umwandlung der Anwendung in JavaScript
ChameRIA	Windows, Linux, Mac	möglich	Impl. als Weban-wendung, z.B. mit JSF	sicher	Umwandlung in eine Webanwen-dung

Tabelle 2.1.: Übersicht zur Bewertung verwandter Arbeiten

Die aufgeführten Arbeiten lassen sich in zwei Gruppen teilen:

Die erste Gruppe nutzt den Browser selbst als Plattform. Hierzu gehören Adobe Flex, Mozilla XUL und PhoneGap. Die Anwendung selbst muss hier mit den Möglichkeiten des Browsers auskommen und das setzt eine Implementierung der Anwendung oder ihrer Oberfläche in JavaScript oder als Web-Anwendung voraus. ChameRIA nutzt den Browser zwar nicht direkt als Plattform, sondern bettet ihn ein. Die Präsentationslogik läuft allerdings innerhalb des Browsers ab, womit sich ChameRIA auf JavaScript für die Implementierungstechnologie beschränkt und damit für die Präsentationslogik in die erste Gruppe einzuordnen ist.

In der zweiten Gruppe, nämlich JavaFX, WPF und SWT, wird der Browser als ein Baustein zur Einbettung betrachtet. Damit kann eine mit Web-Oberflächen-Technologien erstellte Oberfläche aus einer Desktop-Anwendung heraus dargestellt werden. Da der Browser ein untergeordneter Baustein in der jeweiligen Laufzeitumgebung ist, stehen der Desktop-Anwendung weiterhin alle Möglichkeiten zur Integration mit der Plattform zur Verfügung.

Durch die Beschränkung der WPF auf die .Net-Plattform und die Nähe von JavaFX bzw. SWT zur grundlegenden Technologie des vorliegenden Softwaresystem, nämlich Java, sind die zu erwartenden Integrationsaufwände bei JavaFX und SWT im Vergleich zu WPF als wesentlich niedriger einzuschätzen.

JavaFX liefert mit `WebView` und `WebEngine` eine einfache Möglichkeit zur Integration des Browsers mit. Allerdings ist JavaFX 2.x vergleichweise jung und unausgereift[4] und auch die Akzeptanz des Produktes und damit die Beständigkeit der Technologie ist aufgrund der niedrigen Akzeptanz von JavaFX 1.x fraglich.

SWT dagegen ist bereits Teil der vorliegenden Softwareanwendung. Die Einbettung des Browsers in die Desktop-Anwendung ist wie bei JavaFX ebenfalls möglich. Eine Java-Abbildung des Browser-DOMs, wie JavaFX sie mit der `WebEngine` liefert, liegt zwar nicht direkt vor, kann aber über die zur Verfügung gestellten JavaScript-Schnittstellen entwickelt werden.

So stellt SWT eine vernünftige Ausgangsbasis für die weitere Analyse und den Aufbau eines Prototyps dar.

2.3. Zusammenfassung

Die Auswertung verwandter Arbeiten hat mehrere Technologien und Arbeiten gezeigt, die sich mit dem Thema Web-Oberflächen-Technologie in Desktop-Anwendungen befassen.

[4] Bei der Erstellung des Prototypen zeigten sich u.a. Darstellungsfehler und Probleme beim Scroll-Verhalten im Zusammenhang mit der Integration mit SWT.

Der Funktionsumfang dabei reicht von der einfachen Anzeige von Web-Seiten in Desktop-Anwendungen bis zur Bereitstellung von kompletten Anwendungsplattformen. Keine der untersuchten Arbeiten versuchte bisher den eingebetteten Browser bei Erhalt von Java für die Entwicklung der Präsentations- und Geschäftslogik zu nutzen. Allerdings finden sich Technologien, die dies ermöglichen. Eine davon ist das Standard Widget Toolkit (SWT), das die Einbettung des Browsers in eine Java-Desktop-Anwendung ermöglicht.

Auf Basis von SWT werden in der folgenden Analyse die funktionalen Anforderungen an ein Rahmenwerk aufgestellt, um Desktop-Anwendungen mit Web-Oberflächen-Technologien unter Erhalt der Präsentationslogik in Java erstellen zu können.

3. Analyse

Wie in der Einleitung beschrieben, soll die Oberfläche einer vorliegenden Desktop-Anwendung mit eingebettetem Browser unter Verwendung der Web-Oberflächen-Technologien HTML und CSS entwickelt werden. Die Steuerung der Oberfläche, die Präsentationslogik (vgl. Definition in Abschnitt 3.1), soll jedoch weiterhin in Java realisiert werden.

In Abschnitt 3.1 wird hierfür mit dem Model-View-Presenter-Muster eine Architekturvorlage beschrieben, um die Präsentationslogik von der Oberflächen-Technologie unabhängig zu halten.

Im Anschluss werden in Abschnitt 3.2 die funktionalen Anforderungen herausgearbeitet, die ein Rahmenwerk erfüllen muss, um die Entwicklung der Oberfläche einer Desktop-Anwendung mit eingebettetem Browser zu ermöglichen. Diese Anforderungen werden am Beispiel des /Beleg-Editors hergeleitet.

Die nicht-funktionalen Anforderungen an das Rahmenwerk werden in Abschnitt 3.3 aufgestellt.

3.1. Architekturvorlage für die Präsentationsschicht

Informationssysteme werden im allgemeinen aus Schichten konstruiert (vgl. Balzert 1999, S. 372). Bei Systemen mit grafischen Oberflächen unterscheidet man u.a zwei wesentliche Schichten: Die Präsentations- und die Geschäftsschicht (vgl. Abbildung 3.1). Jede der beiden Schichten übernimmt Teilfunktionalitäten, die im Folgenden für diese Arbeit definiert und voneinander abgegrenzt werden.

Als Geschäftsschicht bezeichnet man den Teil eines Softwaresystems, der die fachlichen Konzepte aus dem Geschäftsfeld in Software realisiert. Hier liegen Datenstrukturen vor, die Gegenstände oder Begriffe aus der Geschäftswelt repräsentieren (z.B. Personen, Verträge oder ihre Beziehungen miteinander). Die Operationen auf den Geschäftsdaten (z.B. Vertragsabschlüsse) oder auch Prozesse (z.B. Bezahlvorgänge als ein gemeinsamer Ablauf mit einem Geldinstitut), die aus dem Geschäftsfeld bekannt sind oder abgeleitet werden können, bezeichnet man als Geschäftslogik. Die aus Geschäftsdaten und -logik bestehende

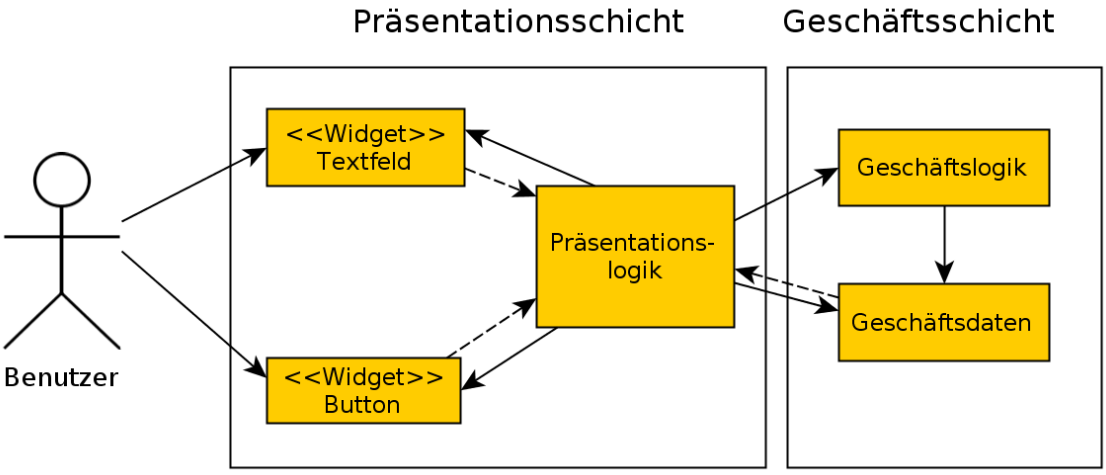


Abbildung 3.1.: Schichtenarchitektur eines Softwaresystems mit Oberfläche

Geschäftsschicht ist für den Benutzer von außen nur indirekt, nämlich nur über die Oberfläche, wahrnehmbar.

Die Präsentationsschicht dagegen realisiert die Darstellung des inneren Zustands eines Softwaresystems. Sie besteht aus Datenstrukturen, die beschreiben, welche Ausgabe- und Eingabeelemente, im folgenden Widget[1] genannt, dem Benutzer zur Verfügung stehen sollen, z.B. in Form von HTML. Der zweite Teil der Präsentationsschicht ist die Präsentationslogik, die im folgenden für diese Arbeit definiert wird:

Präsentationslogik: Die Präsentationslogik definiert das Verhalten einer grafischen Benutzer-Oberfläche, indem es die dargestellten Widgets in ihren Eigenschaften verändert (z.B. kann bearbeitet werden, hat ungültigen Inhalt), neue Widgets hinzufügt oder entfernt. Die Präsentationslogik bedient sich der Geschäftsdaten und Geschäftslogik, um dem Benutzer die fachlichen Gegenstände und Begriffe sowie deren Verhalten und Operationen darauf zugänglich zu machen. Die Präsentationslogik definiert also das Verhalten einer speziellen Oberfläche und grenzt sich auf der einen Seite von den Geschäftsdaten und der Geschäftslogik und auf der anderen Seite von der eingesetzten Oberflächen-Technologie (z.B. SWT oder HTML/CSS) ab, die die Konstruktion von Oberflächen nur im allgemeinen unterstützt.

Die Entwicklung der Präsentationslogik stellt einen erheblichen Anteil am Gesamtentwicklungsaufwand einer Oberfläche dar, da sie individuell für eine Oberfläche erstellt werden

[1] Widget: Grafisches Element eines Oberflächen-Toolkits, z.B. ein Ausgabeelement wie Label oder Eingabeelemente wie Textfeld oder Button.

muss und das wesentliche Verhalten der Oberfläche beinhaltet, das eine erhebliche fachliche Komplexität aufweisen kann. Daraus erhält die Präsentationslogik innerhalb der Präsentationsschicht einen besonderen Stellenwert.

Trotz ihrer Wichtigkeit und ihres hohen Entwicklungsaufwands wird die Präsentationslogik oft nur unzureichend getestet. Oberflächen-Tests[2] sind sehr aufwändig und da die Präsentationslogik meistens eng mit der Oberflächen-Technologie verwoben ist, wird dann nur manuell getestet. Die Trennung von Präsentationslogik von der verwendeten Oberflächen-Technologie verbessert die Testbarkeit der Präsentationslogik.

Für die geplante Modernisierung der vorliegenden Software, bei der die Oberflächen-Technologie durch die Web-Oberflächen-Technologie ergänzt oder ersetzt werden soll, stellt diese Trennung zudem einen möglichen Migrationspfad in Aussicht, weil die Präsentationslogik trotz Austauschs der Oberflächen-Technologie ohne Änderung beibehalten werden kann.

Ein möglicher Migrationspfad für das vorliegende Softwaresystem wäre also, im ersten Schritt die Präsentationslogik aus den heutigen Views herauszulösen und im Folgenden die Oberflächen-Technologie unter Beibehaltung der Präsentationslogik zu ersetzen. Ein Architekturmuster, das diese Trennung von Oberflächen-Technologie und Präsentationslogik propagiert, ist das Model-View-Presenter-Muster, das im folgenden erklärt wird.

3.1.1. Model-View-Presenter (MVP)

Ziel des Model-View-Presenter-Musters (MVP) ist ähnlich zum weitläufig bekannten Model-View-Controller-Muster (vgl. Ezra 2008) die Zerlegung der verschiedenen Bestandteile einer Oberfläche in Bausteine mit jeweils einzelnen Verantwortlichkeiten. Model, View und Presenter werden im folgenden beschrieben:

Das Model repräsentiert die Geschäftsdaten und Geschäftslogik einer Anwendung. Der Presenter realisiert die Präsentationslogik. Er befindet zwischen Model und View, steht mit beiden in direkter Beziehung und vermittelt zwischen ihnen.[3] Aufgabe der View ist die Repräsentation einer Teil-Oberfläche, die sich aus einzelnen Widgets zusammensetzt (vgl. „View is a composition of widgets.“, Ezra 2008). Die View sondiert und verändert die angezeigte Oberfläche über direkten Zugriff auf die Widgets aus der Oberflächen-Technologie. Eingaben

[2] z.B. automatisiert über Robots oder sogar manuelle Tests

[3] Dies ist allerdings nur eine mögliche Form des MVP. Sie wird von Fowler als PassiveView (Fowler 2006a) bezeichnet, da die View sich passiv verhält und darauf 'wartet' vom Presenter mit Daten versorgt zu werden. Eine andere Variante wäre die des Supervising Controller (Fowler 2006b), bei der die View selbständig benötigte Daten durch Direktzugriff auf das Model anfragt. Die Entscheidung für das eine oder das andere ist letztendlich eine Geschmackssache, denn beide Varianten haben ihre Vor- und Nachteile. Da dies für den vorliegenden Sachverhalt nicht wesentlich ist, wird es hier nicht weiter diskutiert.

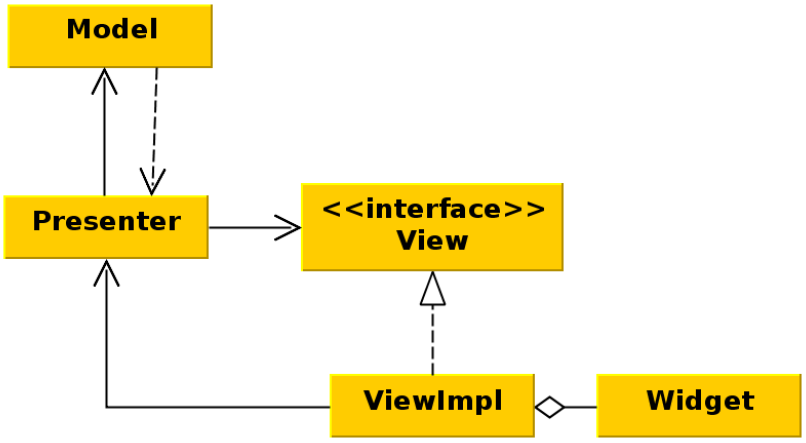


Abbildung 3.2.: Model-View-Presenter

des Benutzers nimmt sie entgegen, indem sie Observer an den Widgets des Oberflächen-Toolkits[4] registriert und bei Benachrichtigung über Ereignisse diese an den Presenter weiterleitet.

Realisiert man nun eine Schnittstelle zwischen Presenter und View, die frei von Eigenschaften der Oberflächen-Technologie ist, erhält man einen Presenter, der die Präsentationslogik bündelt und keine Annahmen über die verwendete Oberflächen-Technologie trifft. Dies erhöht die Testbarkeit der Präsentationslogik und macht den Presenter unabhängig von der Oberflächen-Technologie verwendbar.

3.2. Funktionale Anforderungen

Die Realisierung der Präsentationslogik in Java ist bei der Erstellung von Web-Anwendungen nichts Neues. Die Präsentationslogik wird bei klassischen Web-Server-Anwendungen auf Anfrage durch den Browser serverseitig ausgeführt, worauf die neue Seite als Antwort an den Browser zurückgesendet wird. Dies hat jedoch den Nachteil, dass die Oberfläche aus Sicht des Benutzers nicht besonders reaktiv ist und sich immer im Ganzen verändert. Im Gegensatz dazu kann bei der Verwendung von Oberflächen-Toolkits auf dem Desktop auf feingranulare Ereignisse reagiert und die Oberfläche auch sehr gezielt aktualisiert und verändert werden. Moderne Web-Anwendungen (vgl. AJAX (Wikipedia AJAX 2012)) weichen von dem klassischen Request-Response-Modell ab und werden auch nach dem Ereignismodell realisiert. Hier wird die Präsentationslogik allerdings in JavaScript realisiert und läuft

[4] Oberflächen-Toolkit: Bibliothek zur Erstellung von grafischen Benutzer-Oberflächen, die aus einer Sammlung an Oberflächen-Komponenten, Widgets, und Hilfsklassen zur Oberflächen-Entwicklung besteht.

client-seitig im Browser ab. Aus JavaScript heraus wird auf dem Browser-DOM operiert, um die dargestellten Widgets anzusprechen.

Wird ein Browser nun in die Desktop-Anwendung eingebettet und stellt man dem Oberflächen-Entwickler nun das Browser-DOM im Java-Raum zur Verfügung, kann er darauf wie auf einem Oberflächen-Toolkit operieren. Da es sich hier um eine zu einer einzelnen Anwendung orthogonale Funktionalität handelt, soll diese als Rahmenwerk zur Verfügung gestellt werden. Die funktionalen Anforderungen an das Rahmenwerk werden im Folgenden anhand eines (reduzierten) Beispiels aus der vorliegenden Anwendung, dem /Beleg-Editor (Abbildung 3.3), abgeleitet.

3.2.1. Anwendungsbeispiel /Beleg-Editor

Der /Beleg-Editor (Abbildung 3.3) ist ein Teil der Schadensfallbearbeitung und dient der Neuerfassung und Bearbeitung von Belegen.

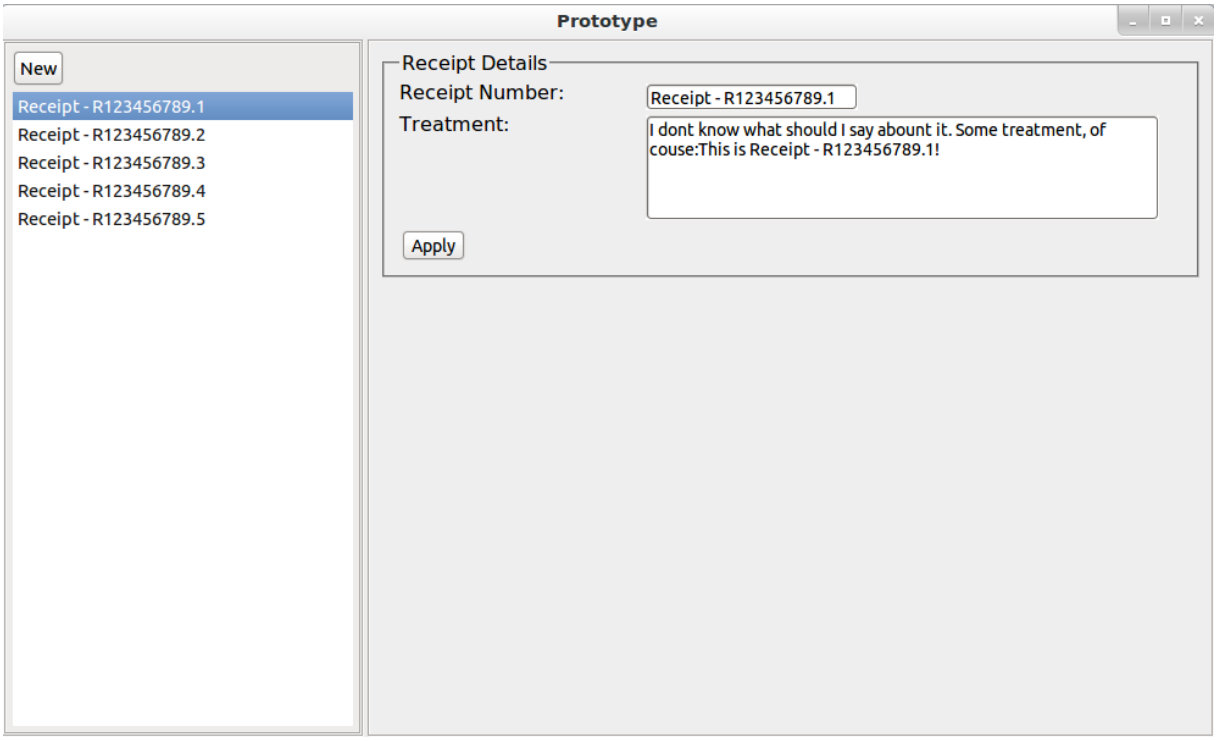


Abbildung 3.3.: Oberfläche des /Beleg-Editors

Der /Beleg-Editor setzt sich aus zwei /Unter-Editoren zusammen (Abbildung 3.4). Beide haben sie keine Kenntnis von der Existenz des anderen und werden über den /Beleg-Editor, bzw. sein Model, koordiniert.

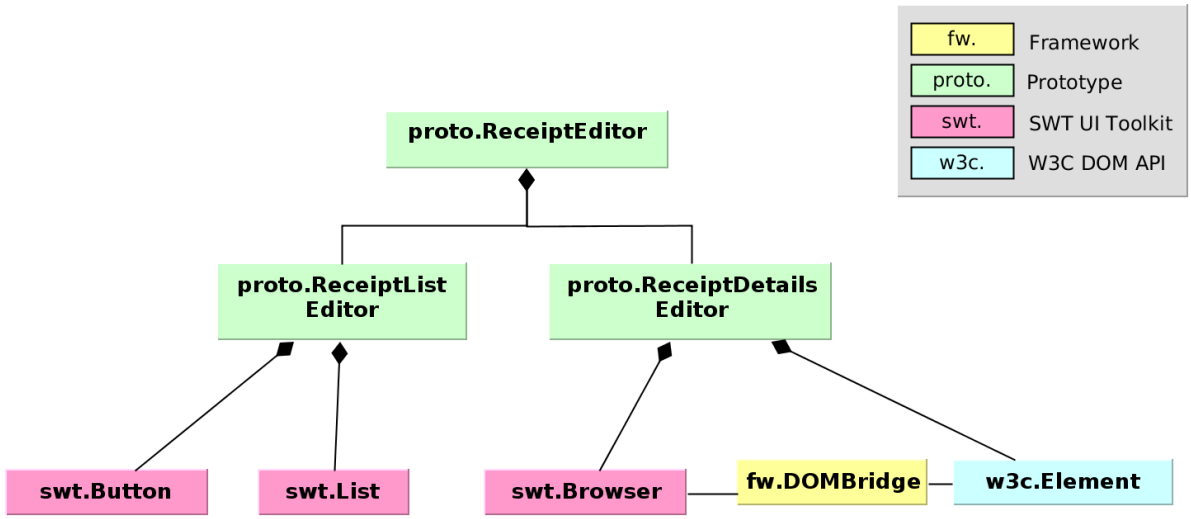


Abbildung 3.4.: Aufbau des /Beleg-Editors

Der Editor auf der linken Seite (s. Abbildung 3.3), im Folgenden ReceiptListEditor genannt, zeigt eine Liste an Belegen an. Über Klick auf die Elemente dieser Liste können Belege ausgewählt werden. Durch Auswählen des Neu-Button über der Liste kann ein neuer Beleg erzeugt werden.

Der rechte Editor (s. Abbildung 3.3), der ReceiptDetailsEditor, ist ein Formular, das der Bearbeitung der Beleg-Detail-Daten dient. Die Daten des gewählten Belegs werden nach Auswahl seines Eintrags in der Liste in das Formular eingefüllt und können dort bearbeitet werden. Veränderte Daten aus dem Formular werden übernommen, wenn dies durch Klick auf den Übernehmen-Knopf angefordert wird.

Zur Ableitung der Anforderungen an das Rahmenwerk soll Folgendes angenommen werden, um sicherzustellen, dass auch Interaktionen zwischen alter und neuer Oberflächen-Technologie berücksichtigt werden: Der ReceiptListEditor soll auf der bisher eingesetzten Technologie SWT umgesetzt sein. Der ReceiptDetailsEditor soll mit Einsatz von HTML und CSS realisiert sein. Zu jedem der Editoren soll die Präsentationslogik in Java entwickelt vorliegen.

Ein Anwendungsfall des /Beleg-Editors, der beide Editoren betrifft, ist die Auswahl und Bearbeitung eines Belegs. Der Ablauf ist im Kollaborationsdiagramm Abbildung 3.5 beschrieben:

Initiiert wird der Ablauf durch Klick auf einen Eintrag in der Liste durch den Benutzer (s. Schritt 1.1). Der ReceiptListEditor wird hierüber in Schritt 1.2 vom Widget seines Oberflächen-Toolkits über dieses Ereignis informiert. Dieser weist das Model in Schritt 1.3 nun an, den

gewünschten Beleg zu selektieren. Das ReceiptModel hat nun die Wahl, ob es dieser Aufforderung folgt. Im vorliegenden Beispiel entscheidet das ReceiptModel, den Beleg zum aktuell Selektierten zu machen (Schritt 1.4). Darüber informiert das ReceiptModel nun alle Beobachter. Als erstes ist dies der ReceiptListEditor selbst (Schritt 1.5), der die Selektion des Belegs im Widget des Oberflächen-Toolkit in Schritt 1.6 für den Benutzer sichtbar macht. Mit Schritt 1.7 informiert das ReceiptModel den zweiten Beobachter, den ReceiptDetailsEditor. Dieser muss nun, um die Selektion des Belegs nachzuvollziehen, die Daten des aktuellen Belegs aus dem ReceiptModel abfragen und in die Formularfelder des Browser einfüllen. Dies geschieht in Schritt 1.8.

In Schritt 2.1 verändert der Benutzer den Inhalt eines Feldes. Dieses Ereignis wird dem ReceiptDetailsEditor mitgeteilt (Schritt 2.2). Nun kann der ReceiptDetailsEditor in Schritt 2.3 den Inhalt des veränderten Feldes auslesen und validieren. (Die Inhalte der Felder sind gültig, darum unternimmt der ReceiptDetailsEditor nichts.)

Durch Klick auf den Apply-Button (Schritt 3.1) fordert der Benutzer an, die Änderung in den gewählten Beleg zu übernehmen. Das DOM des Browsers informiert nun den ReceiptDetailsEditor in Schritt 3.2 über das Ereignis. Der ReceiptDetailsEditor weist das ReceiptModel an, den Beleg mit den Daten aus dem Formular zu aktualisieren (Schritt 3.3). Dies führt das ReceiptModel in Schritt 3.4 durch und informiert wiederum seine Beobachter. Dies erfolgt analog zu Schritt 1.5 und 1.7. Damit ist der Anwendungsfall abgeschlossen.

Die einzelnen Schritte werden in den folgenden Abschnitten zur Ableitung der funktionalen Anforderungen herangezogen.

3.2.2. Ressourcen-Bereitstellung

In den verwandten Arbeiten hat sich gezeigt, dass sich mit Einsatz des Standard Widget Toolkit ein Browser in die Java-Anwendung einbetten und innerhalb der Anwendungsoberfläche anzeigen lässt.

Damit der eingebettete Browser die Oberfläche des ReceiptDetailsEditors darstellen kann, benötigt er eine Beschreibung der Oberfläche, auf deren Basis er den DOM-Baum zur Darstellung aufbauen kann. Die Beschreibung liegt als HTML-Datei z.B. im Java-Klassenpfad der Anwendung vor. Es kann aber auch sein, dass diese HTML-Datei aus dem Dateisystem oder von einem externen System, z.B. aus einer Datenbank oder einem Content-Management-System geladen werden soll. Zusätzlich zu HTML-Dateien können auch andere Formate wie Stylesheets in Form von CSS-Dateien, Skripte, Bilder oder andere Medienformate, wie Flash-Filme abgerufen werden. Die Ressourcen-Bereitstellung soll für den Oberflächen-Entwickler transparent, aber erweiterbar, erfolgen.

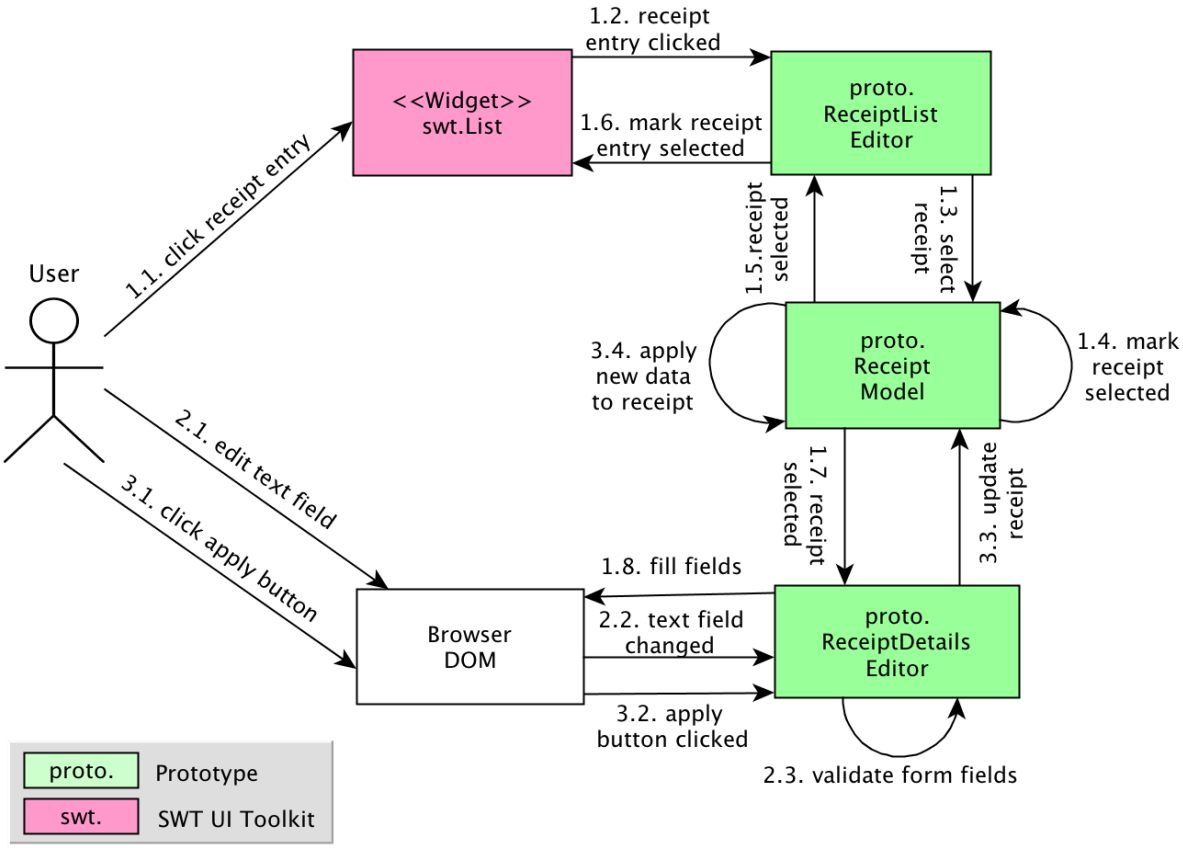


Abbildung 3.5.: Kollaborationsdiagramm des /Beleg-Editor-Beispiels

Anforderung: Die hier zu entwickelnde Rahmenwerkskomponente, die Ressourcen-Bridge, soll die vom Browser angeforderten Ressourcen bei der Anwendung erfragen und an den Browser ausliefern. Allerdings soll die Rahmenwerkskomponente nicht eng an die Realisierung der Ressourcen-Bereitstellung gekoppelt sein, da es die Wiederverwendbarkeit des Rahmenwerks zunichte machen würde.

3.2.3. Sondierung und Veränderung des Browser-DOMs

Die für die Arbeit mit dem eingebetteten Browser relevanten Schritte beginnen im dargestellten Beispiel 3.2.1 mit Schritt 1.8. Hier sollen die Daten aus dem selektierten Beleg in das Formular eingefüllt werden. Die Datenquelle ist der aktuell selektierte Beleg des ReceiptModels. Als Datensenke soll nun der Browser bzw. die durch ihn dargestellten Widgets dienen. Diese werden durch das Document Object Model (DOM) innerhalb des Browsers repräsentiert.

Das DOM ist eine Baumstruktur, die aus Knoten, `Nodes`, besteht. Als Rahmen der Gesamtstruktur dient das `Document`. Das `Document` enthält ein `Element`. Ein `Element` ist ein `Node`, der auch wiederum andere `Nodes` (Kindknoten bzw. Child Nodes) enthalten kann. Zudem kann ein `Element` Eigenschaften, `Attributes`, bestehend aus einem Wert, der einem Namen zugeordnet ist (vgl. Diagramm 4.4 im Kapitel Design). Dazu ein einfaches Beispiel:

```
1 <form>
2   <input type="text" id="textField" value="some text" />
3 </form>
```

Listing 3.1: Beispiel eines DOM

Das `Document` aus Beispiel Listing 3.1 hält an der Wurzel das `Element` 'form', welches keine Attribute besitzt. Ein Textfeld wird im Browser-DOM durch ein `Element` vom Typ 'text', s. `Attribute type='text'` in Zeile 2, repräsentiert. Dieses Feld wird durch einen Bezeichner, `id='textField'`, im gesamten `Document` eindeutig identifiziert. Der aktuelle Wert des Textfeldes wird definiert durch `value='some text'`.

Zurück zum Beispiel: Soll in Schritt 8 das Formular befüllt werden, müssen hierzu die vom eingebetteten Browser dargestellten Widgets aus dem DOM ermittelt werden. Dies kann z.B. anhand der in Beispiel Listing 3.1, Zeile 2 gezeigten Bezeichner erfolgen. Der Inhalt des ermittelten Widgets kann nun über die Veränderung des `Attribute value` bestimmt werden. Dies ist ein Beispiel für einen Element-Typ. Das Browser-DOM ist genau so konzipiert, dass (fast) alle Änderungen an Widgets durch das Verändern von `Element`-Knoten oder deren `Attributes` durchgeführt werden können.

Weiter im Beispiel fällt in Listing 3.5 Schritt 10 auf, dass die Präsentationslogik Validierungen durchführt. Hierzu ist es notwendig, dass der Wert eines Widgets auch gelesen werden kann. Dieses findet analog zum Beschreiben der `Attributes` als Auslesen des Werts eines `Attributes` statt.

Hieraus ergeben sich folgende Anforderungen:

Anforderung: Da das Browser-DOM nativ im Browser enthalten ist und die Programmierung in Java durchgeführt werden soll, muss eine Rahmenwerkskomponente erstellt werden, die die Lokalisierung von Elementen des Browser-DOM realisiert.

Anforderung: Der Anwendungs-Entwickler muss auf den lokalisierten Elementen Veränderungen an dessen Attributen durchführen können.

Anforderung: Der Anwendungs-Entwickler muss die Attribute von lokalisierten Elementen auslesen können.

3.2.4. Ereignisbenachrichtigung des Browser-DOMs

Die Befüllung der Formularfelder mit den Daten des aktuellen Belegs ist in Abbildung 3.5 Schritt 8 abgeschlossen. In Schritt 9 führt der Benutzer die Änderung am Inhalt eines Textfeldes durch. Damit die Präsentationslogik des ReceiptDetailsEditors jetzt Validierungen durchführen kann, muss er über die Änderung informiert werden. Dies erfolgt über Ereignisse, die vom DOM ausgesandt werden. Für Ereignisse eines Typs, z.B. onchange oder onclick, kann man sich am nativen Browser-DOM mit einem Callback, dem Event-Listener, registrieren, so dass man bei ihrem Auftreten darüber informiert wird. Einen solchen Listener muss der ReceiptDetailsEditor am Textfeld, dessen Inhalt durch den Benutzer verändert wird, beim Eintreten des Ereignisse bereits angemeldet haben.

Aus diesem Beispiel ergibt sich folgende Anforderung:

Anforderung: Der Anwendungs-Entwickler muss an DOM-Elemente einen Event-Listener mit eigener Implementierung registrieren können, so dass während der Aktivierung Informationen über das Ereignis, z.B. die Eingabe des Benutzers, zur Verfügung stehen.

3.3. Nichtfunktionale Anforderungen

3.3.1. Erweiterbarkeit

Ein Rahmenwerk soll sich weiter entwickeln können, auch nachdem es bereits in der Benutzung ist. Dies soll erreicht werden durch eine durchgängige Kapselung der Implementierungsdetails und dem Vorsehen von Erweiterungspunkten. So soll im vorliegenden Fall der Typ des eingebetteten Browser oder wie sein DOM angesprochen wird für den Oberflächen-Entwickler nicht relevant sein.

3.3.2. Portabilität

Da die untersuchte Software unter Windows und Linux lauffähig ist, sollte das Rahmenwerk diese Plattformen auch unterstützen.

3.4. Zusammenfassung

In diesem Kapitel wurde eine Architekturvorlage erarbeitet, die es erlaubt die Präsentationsschicht so zu strukturieren, dass die Präsentationslogik von der Oberflächen-Technologie getrennt vorliegt. Damit ist die Oberflächen-Technologie austauschbar ohne Änderungen an der Präsentationslogik nach sich zu ziehen. Diese Architekturvorlage stellt einen wichtigen Zwischenschritt bei der Migration des vorliegenden Softwaresystems dar. Entwicklungen von neuen Oberflächen-Teilen sollen ebenfalls diesem Aufbau folgen.

Anhand eines fachlichen Beispiels, das im Entwurf und in der Realisierung als Grundlage für den Prototyp fungieren wird, wurden die funktionalen Anforderungen an das Rahmenwerk aufgestellt. Zur Entwicklung der Oberfläche einer Desktop-Anwendung unter Einsatz der Web-Oberflächen-Technologien mit eingebettetem Browser haben sich zwei Komponenten als wesentlich herausgestellt:

- Eine Ressource-Bridge soll die Versorgung des eingebetteten Browsers mit zur Darstellung der Oberfläche benötigten Ressourcen sicherstellen.
- Die DOM-Bridge soll den Zugriff auf das native Browser-DOM über Java-Proxies ermöglichen, um dieses wie ein herkömmliches Oberflächen-Toolkit einsetzen zu können. Die DOM-Bridge muss die Lokalisierung von DOM-Elementen erlauben.
- Die lokalisierten DOM-Elemente müssen in ihren Eigenschaften verändert werden können.
- Die Eigenschaften von DOM-Elementen müssen ausgelesen werden können.
- Der Empfang von Ereignissen aus dem Browser-DOM muss möglich sein.

Als nicht-funktionale Anforderungen an das Rahmenwerk wurde Erweiterbarkeit und Portabilität ermittelt:

- Die Erweiterbarkeit des Rahmenwerks um anwendungsspezifisches Verhalten soll bereits über die Komponente Ressourcen-Bridge erreicht werden.
- Die Portabilität würde dadurch erreicht, dass das Rahmenwerk wie das untersuchte Softwaresystem auf den Plattformen Linux und Windows lauffähig und in seinen Funktionen identisch ist.

Im folgenden Abschnitt wird ein Entwurf für die Rahmenwerks-Komponenten Ressourcen-Bridge und DOM-Bridge erstellt. Die Anwendbarkeit des Rahmenwerks soll über einen Prototypen evaluiert werden, der die MVP-Architekturvorlage berücksichtigen soll.

4. Design

Ziel dieses Kapitels ist die Erstellung eines Entwurfs für das Rahmenwerk und den Prototypen.

Die Rahmenwerks-Komponente Ressource-Bridge soll den Browser mit den für die Darstellung benötigten Ressourcen versorgen. Die Herkunft dieser Ressourcen soll vom Rahmenwerks-Nutzer festgelegt werden können. Da das Rahmenwerk keine Annahmen über die Anwendung treffen soll, um davon unabhängig zu bleiben, muss die Ressource-Bridge so entworfen werden, dass sie um verschiedene Strategien zur Ressourcen-Versorgung erweitert werden kann.

Für die Arbeit mit dem eingebetteten Browser wird der Zugriff auf die dargestellten Widgets benötigt. Diese werden innerhalb des Browsers über sein DOM repräsentiert. Damit Sondierungen und Änderungen an den Oberflächen-Elementen durchgeführt werden können, soll das native Browser-DOM von der Rahmenwerks-Komponente DOM-Bridge zugänglich gemacht werden. Um Ereignisse des Browser-DOMs, z.B. Klicken auf einen Button, in der Java-Welt empfangen und verarbeiten zu können, soll die DOM-Bridge die Registrierung und Benachrichtigung von Java-Event-Listenern ermöglichen.

Der Entwurf des Prototyps soll die Funktionsfähigkeit des Rahmenwerks und die Anwendbarkeit des Architekturvorschlags auf Basis des Model-View-Presenter-Musters nachweisen.

Das Diagramm 4.1 zeigt eine Gesamtübersicht über den Prototypen. Dieser enthält den /Beleg-Editor, wie er bereits in der Analyse beschrieben worden ist, unter Nutzung der Rahmenwerk-Funktionalitäten.

Die Oberfläche des /Beleg-Editors setzt sich aus `ReceiptListEditor` und `ReceiptDetailsEditor` zusammen. Die `ReceiptListEditor` soll die Widgets des SWT nutzen. Dies entspricht einer Realisierung auf Basis der bisher eingesetzten Oberflächen-Technologie.

Der `ReceiptDetailsEditor` dagegen soll auf Basis der Web-Oberflächen-Technologien realisiert werden. Dies erfolgt über die Einbettung des Browsers. Die Entwicklung seiner Präsentationslogik soll weiter in Java erfolgen.

Zur Darstellung muss der Browser mit der Beschreibung der Teil-Oberfläche in Form einer HTML-Beschreibung versorgt werden. Die HTML-Beschreibung und weitere zur Darstellung

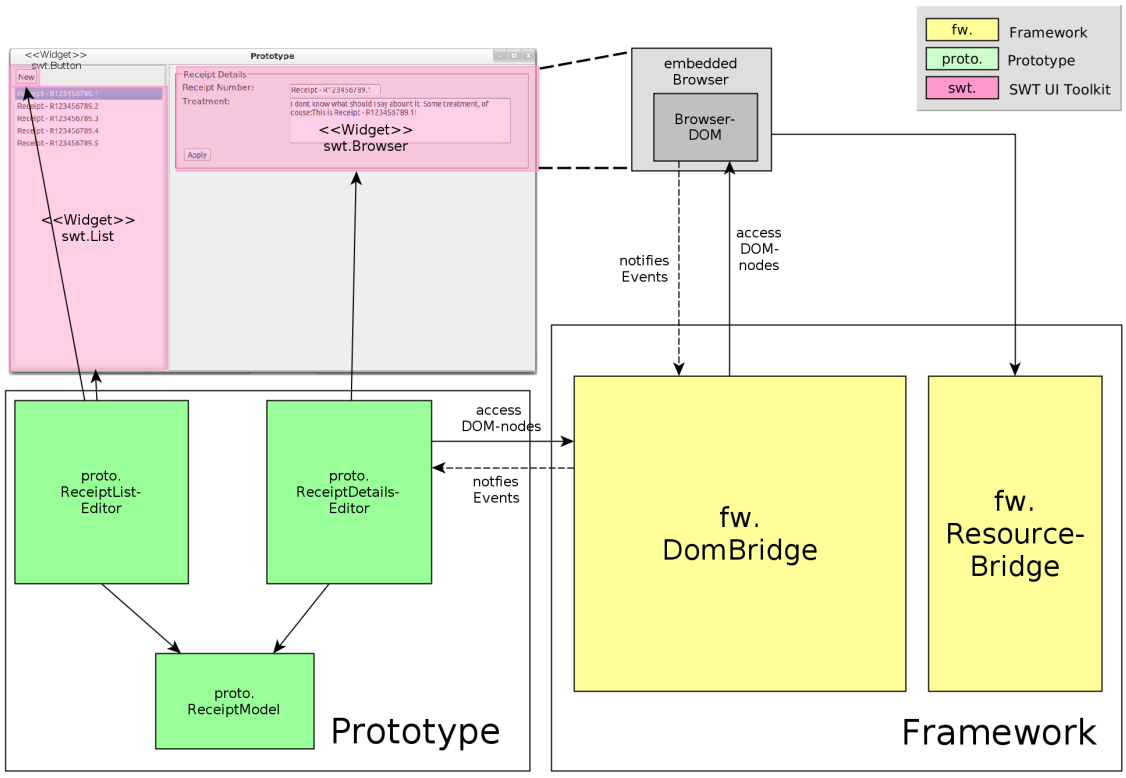


Abbildung 4.1.: Übersicht über Komponenten von Rahmenwerk und Prototyp

benötigte Ressourcen werden dem Browser über die Komponente Ressource-Bridge zur Verfügung gestellt.

Nach der in der Analyse erarbeiteten Architekturvorlage sollen sowohl der `ReceiptList-Editor`, wie auch der `ReceiptDetailsEditor` nach dem Model-View-Presenter-Muster aufgebaut werden. Hierbei beinhaltet der jeweilige Presenter die Präsentationslogik während die dazugehörige View-Implementierung den Zugriff auf die Oberflächen-Technologie kapseln soll. Den Zugriff auf die Oberflächen-Technologie in Form der dargestellten Widgets erhält die `ReceiptListView`, da sie eine reine SWT-Implementierung ist, auch durch die Widgets von SWT. Die `ReceiptDetailsView` dagegen basiert auf den Web-Oberflächen-Technologien und benötigt deshalb Zugriff auf das DOM des Browsers, das die dargestellten Widgets repräsentiert. Zugriff auf dieses DOM erhält die `ReceiptDetailsView` über die DOM-Bridge.

Im Folgenden wird als erstes ein Entwurf für das aus Ressource-Bridge und DOM-Bridge bestehende Rahmenwerk beschrieben. Im zweiten Schritt liefert dieses Kapitel den Entwurf eines Prototyps zur Evaluation, ob die Architekturvorlage und das Rahmenwerk die gestellten Anforderungen erfüllen.

4.1. Rahmenwerk

Das Rahmenwerk besteht aus zwei Kernkomponenten: Ressource-Bridge und DOM-Bridge, die im folgenden erläutert werden.

4.1.1. Ressource-Bridge

Aufgabe der `ResourceBridge` (s. Klassendiagramm Abbildung 4.2) ist die Versorgung des eingebetteten Browsers mit Ressourcen. Auf welche Art dies erfolgt soll für den Rahmenwerk-Anwender nicht sichtbar sein. Die Herkunft der Ressourcen soll er allerdings selbst bestimmen können. Dies kann er durch die Implementierung einer eigenen Strategie erreichen, die das Interface `ResourceProvider` (s. 4.2) erfüllt, und an der `ResourceBridge` registriert wird.

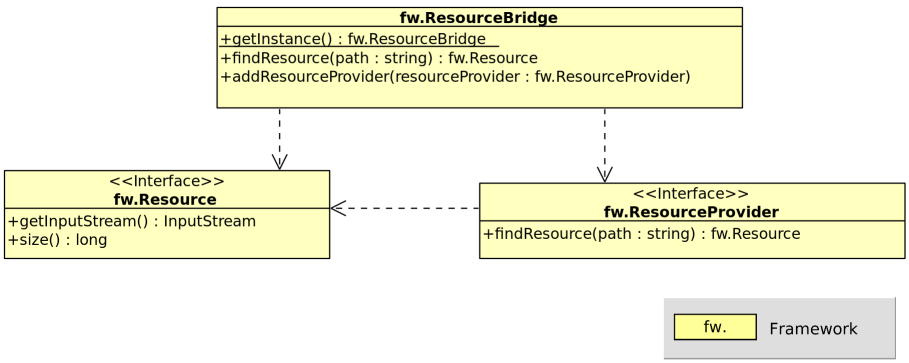


Abbildung 4.2.: Design der Ressource-Bridge

Die `ResourceBridge` muss vom Rahmenwerk-Nutzer initial mit mindestens einem `ResourceProvider` ausgestattet werden, ansonsten können keine Ressourcen ausgeliefert werden. Dazu gibt es z.B. den `ClasspathResourceProvider`, der Ressourcen aus dem Java-Klassenpfad bereitstellt. Sind mehrere `ResourceProvider` vorhanden, so probiert die `ResourceBridge` so lange die verschiedenen `ResourceProvider`-Strategien durch, bis eine angeforderte `Resource` geliefert wurde.

Zur Anbindung der `ResourceBridge` an den Browser sind mehrere Vorgehensweisen möglich. Eine naheliegende Realisierung wäre das Ausliefern der Ressourcen aus dem Dateisystem. Der eingebettete Browser kann nativ[1] auf das Dateisystem zugreifen. Dagegen spricht allerdings, dass eine als Jar-Dateien paketierte Software mit all ihren Ressourcen in das Dateisystem ausgepackt oder alle Ressourcen initial in das Dateisystem exportiert (und

[1] Über das file://-Protokoll

später wieder gelöscht) werden müssten. Eine zweite Möglichkeit wäre die Auslieferung der Ressourcen über eine lokale TCP/IP-Socket, die durch einen leichtgewichtigen eingebetteten HTTP-Server bedient wird. Diese Integration entspricht dem üblichen Kommunikations-Modell mit dem Browser und ist deshalb leicht anwendbar.[2] Als dritte Möglichkeit besteht die Anbindung an eine Browser-spezifische Schnittstelle. Der vorliegende Browser WebKit bietet hierbei die WebKit-WebResource-API (WebKit Source Code 2012) an, die explizit für diesen Zweck vorgesehen ist.

Für eine prototypische Implementierung eignet sich die Vorgehensweise mit dem eingebetteten HTTP-Server auf lokaler TCP/IP-Socket am besten, da sie leicht realisierbar ist. Optimal wäre die Anbindung an die Browser-spezifische Schnittstelle, da so eine transparente Ressourcen-Auslieferung ermöglicht wird, die nicht über den Umweg des TCP/IP-Stacks geht. Da der Aspekt, wie die Versorgung des Browsers tatsächlich erfolgt, vor dem Rahmenwerk-Anwender verborgen ist, soll dies im Rahmen des Prototyps erst einmal über einen einfachen eingebetteten HTTP-Server über lokale TCP/IP-Socket erfolgen. Dies wird in der Realisierung näher betrachtet (vgl. Kapitel 5).

4.1.2. DOM-Bridge

Die zweite wesentliche Komponente des Rahmenwerks ist die DOM-Bridge. Aufgabe der DOM-Bridge ist es, dem Anwendungsentwickler Zugriff auf das DOM des Browsers zu gewähren. Da das W3C-spezifizierte DOM-API aus dem Java Runtime Environment eine standardisierte Schnittstelle darstellt, die den meisten Java-Entwicklern aus der Verarbeitung von XML bekannt sein sollte und die auch dem internen Browser-DOM sehr ähnlich ist, soll die DOM-Bridge diese Schnittstelle bereitstellen und die Lücke zwischen Java-Objekt-Raum und Browser-internem DOM überbrücken. Auf diese Weise wird der Code der View-Implementierungen zusätzlich nicht von den Schnittstellen des Rahmenwerks abhängig.

Für jede Instanz des eingebetteten Browsers muss vom Anwendungsentwickler eine eigenständige `DomBridge` instantiiert werden. Dies erfolgt in der View-Implementierung. Da der Browser über das SWT-Widget `Browser` eingebettet wird und dieses auch die Mechanismen zur Kommunikation mit der nativen Browser-Instanz bereitstellt, muss die `DomBridge` mit einer Referenz auf dieses Widget erzeugt werden.

Das DOM (s. Diagramm 4.3) ist als Baum strukturiert und bezeichnet alle seine Bestandteile als `Nodes`. Wie bereits in der Analyse herausgearbeitet worden ist, soll der Anwendungsentwickler Zugriff auf die einzelnen `Nodes` des nativen DOMs durch Stellvertreter-Objekte (Proxy) im Java-Objekt-Raum erhalten. Die Proxy-Objekte müssen demnach alle Methoden

[2] Allerdings soll hiermit explizit nicht das HTTP-Request-Response-Modells für die Realisierung der Oberfläche eingeführt werden.

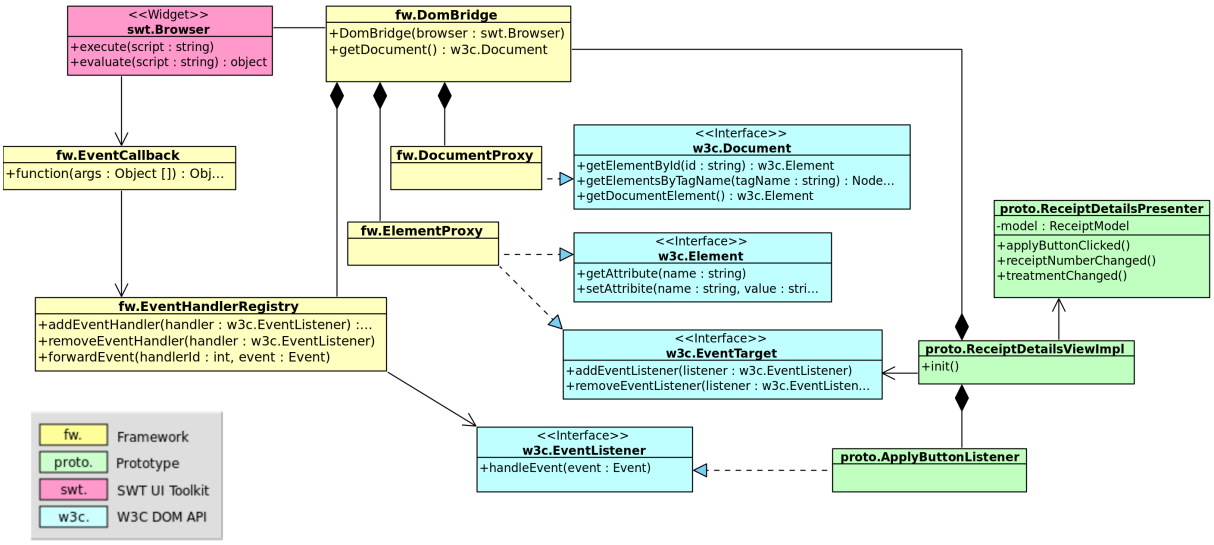


Abbildung 4.3.: Design der DOM-Bridge

der W3C-DOM-API implementieren und durch Abbildung auf das Browser-DOM nachstellen.

Als Wurzel des DOM dient das `Document`-Objekt. Um die Arbeit mit dem DOM beginnen zu können, benötigt der Anwendungsentwickler Zugriff darauf. Diesen erhält er über die `DomBridge`, die über die Methode `getDocument()` einen Proxy auf den nativen Browser-DOM bereitstellt. Die Kommunikation mit dem Browser realisiert der `Document`-Proxy über die `DomBridge`, die wiederum die Referenz auf den eingebetteten Browser über das SWT-Widget `Browser` hält. Ein `Document` besitzt (u.a.) ein Wurzel-Element. Einen Proxy darauf erhält der Anwendungsentwickler vom `Document` bzw. seinem Stellvertreter über die Methode `getDocumentElement()`. Das Wurzel-Element selbst ist ein DOM-`Element`. Ein DOM-`Element` besitzt Attribute und kann wiederum Kind-Knoten (`Nodes`) enthalten.

Über das `Document` können Elemente auch anhand eindeutiger Bezeichner, das Attribut `'id'`, lokalisiert werden. Ebenso kann der Oberflächen-Entwickler beginnend beim `Document` den Baum traversieren. Auch XPath-Abfragen sind auf dem DOM durchführbar, so dass eine Auswahl an Wegen zur Lokalisierung von Elementen zur Verfügung steht.

Hiermit ist die Anforderung an das Rahmenwerk erfüllt, einzelne Knoten des DOM-`Document` lokalisieren zu können.

Da zahlreiche Eigenschaften von Browser-Widgets durch die Attribute von Elementen repräsentiert werden, ist der Umgang mit `Element`-Attributen notwendig. Die Attribute von

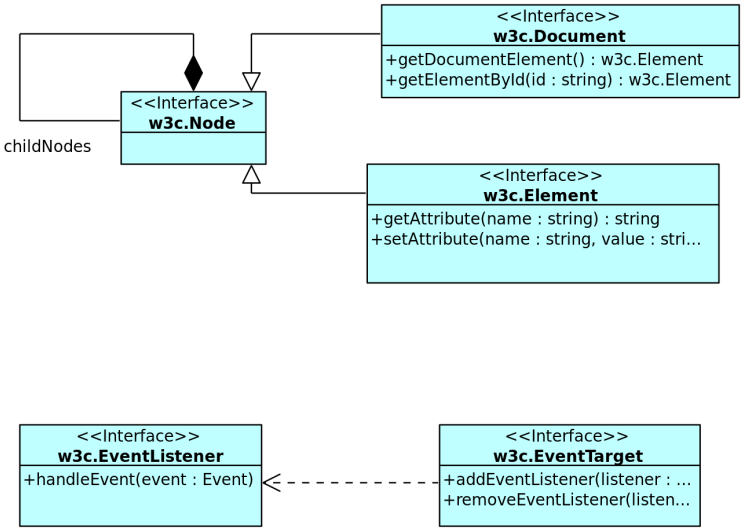


Abbildung 4.4.: Ausschnit aus dem W3C-DOM

Elementen können über die Methoden `getAttribute()` und `setAttribute()` ausgelesen oder verändert werden. Diese werden durch die ElementProxies so implementiert, dass der Wert der über den Attribute-Namen adressierten Eigenschaft im Falle von `getAttribute()` über das Browser-Widget ausgelesen und für `setAttribute()` im Browser-DOM verändert wird.

Damit ist auch die Anforderung, die Attribute von Elementen zu sondieren und verändern zu können, erfüllt.

Teil der W3C-DOM-API in Java ist ein API für die Realisierung von `EventListenern` auf DOM-Elementen (vgl. Diagramm 4.3) Diese API befindet sich im Paket `org.w3c.dom.events`.

Wenn ein Knoten des DOM-Baums die Schnittstelle `EventTarget` implementiert, können an diesen Knoten `EventListener` für Events eines Typs angemeldet werden. Wenn Ereignisse diesen Typs auftreten, wird die Methode `handleEvent()` des Event-Listeners aufgerufen.

Während für das DOM der Proxy im Java-Objekt-Raum existiert und Aufrufe darauf in Aufrufe auf dem nativen Browser-DOM umgesetzt werden, muss dies für Event-Listener in umgekehrter Richtung erfolgen. Hierzu muss eine Abbildung vom nativen Event-Handler auf das als `EventListener` registrierte Java-Objekt durchgeführt werden. Dies wird realisiert über die `EventHandlerRegistry`. Es existiert eine `EventHandlerRegistry`-Instanz pro DOM-Bridge.

Bei Registrierung eines Event-Handlers wird der Java-Event-Handler in einer Tabelle abgelegt und ihm ein Identifizierer zugeordnet. Gleichzeitig wird ein Event-Callback-Objekt, das Kenntnis des Identifizierers hat, an den korrespondierende `Node` im nativen DOM angemeldet, der dann bei Auftreten des Events die DOM-Bridge anweist, das Event an den Event-Handler weiterzuleiten, der diesem Identifizierer zugeordnet ist.

Wie ein vom Benutzer ausgelöstes Ereignis aus dem Browser-DOM bis zum angemeldeten `EventListener` gelangt, zeigt Diagramm Abbildung 4.5 und wird im folgenden beschrieben:

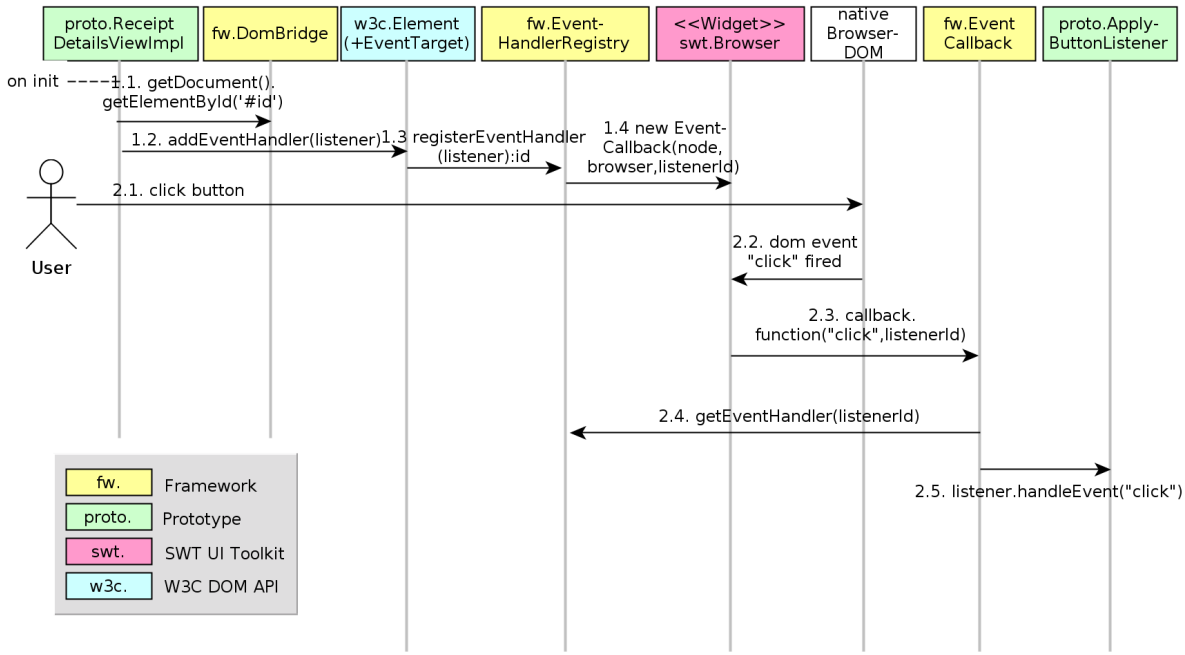


Abbildung 4.5.: Sequenzdiagramm der Ereignis-Verarbeitung vom Browser-DOM bis in den Java-Event-Listener

Bei Initialisierung der Oberfläche lokalisiert die View (s. 4.5, Schritt 1.1 bei 'on init') mit Hilfe der DOM-Bridge über `getDocument().getElementById()` die benötigten DOM-Elemente, z.B. den Übernehmen-Button aus dem /Beleg-Editor-Beispiel, anhand ihrer eindeutigen Bezeichner. Der Apply-Button wird nun durch ein entstprechendes Element-Proxy-Objekt der DOM-Bridge repräsentiert. Dieses implementiert ebenfalls das Interface `EventTarget`, das die Registrierung von Java-Klassen erlaubt, die das `EventListener`-Interface erfüllen. In Schritt 1.2 registriert die `ReceiptDetailsView` den `EventListener ApplyButtonListener` über Aufruf von `addEventListener()` auf dem Element-Proxy. In Schritt 1.3 meldet der Element-Proxy den `ApplyButtonListener` an der `EventHandlerRegistry` der

DOM-Bridge an und erhält dafür einen eindeutigen Bezeichner, die Listener-ID, zurück, über den der `EventListener` innerhalb der `EventHandlerRegistry` eindeutig identifiziert werden kann. Die `EventHandlerRegistry` erzeugt nun mit Schritt 1.4 das Event-Callback-Objekt, das bei Auftreten des gewünschten Events aus dem Browser-DOM heraus aufgerufen wird und aktiviert dieses über das `Browser`-Widget des Oberflächen-Toolkits SWT.

Der Benutzer klickt in Schritt 2.1 auf den Übernehmen-Button. Diese Eingabe führt zu einem Event im nativen Browser-DOM in Schritt 2.2, das in Schritt 2.3 zur Aktivierung des Event-Callbacks führt. Der Event-Callback lokalisiert nun mit Hilfe der Listener-ID den am Element registrierten Event-Listener und erhält den `ApplyButtonListener` (s. Schritt 2.4). In Schritt 2.5 wird die Methode `handleEvent()` auf dem `ApplyButtonListener` vom Event-Callback aufgerufen und die Verarbeitung des Ereignisses kann wie gewohnt in Java weiter erfolgen.

Damit kann an einem Element-Proxy Event-Handler-Objekt angemeldet und für den Empfang von Browser-Events registriert werden. So ist auch die letzte Anforderung an das Rahmenwerk, der Empfang von Ereignissen aus dem nativen Browser-DOM durch ein Java-Objektr, erfüllt.

4.2. Prototyp

Neben der Evaluierung, ob die Entwicklung von Anwendungs-Oberflächen mit dem Rahmenwerk durchgeführt werden kann, dient der Prototyp auch der Überprüfung, ob die auf dem Model-View-Presenter-Muster basierende Architekturvorlage umgesetzt werden kann. Das Klassendiagramm Abbildung 4.6 zeigt den Entwurf des Prototyps, dessen Klassen im Folgenden beschrieben werden.

4.2.1. ReceiptEditor und ReceiptEditorModel

Der `ReceiptEditor` repräsentiert den /Beleg-Editor im ganzen, also den Prototypen an sich und dient als Einstiegspunkt in das Programm.

Als erstes erzeugt der `ReceiptEditor` das Model `ReceiptEditorModel`, das die verfügbaren Belege verwaltet und Änderungen an ihnen überwacht. Die Belege liegen als einfache Liste an Beleg-Objekten im Speicher vor. Für den Prototypen ist ein komplexeres Model der Speicherung nicht notwendig, da dieser Aspekt auch für andere Komponenten transparent ist. Zusätzlich speichert das `ReceiptEditorModel` einen aktuell ausgewählten Beleg.

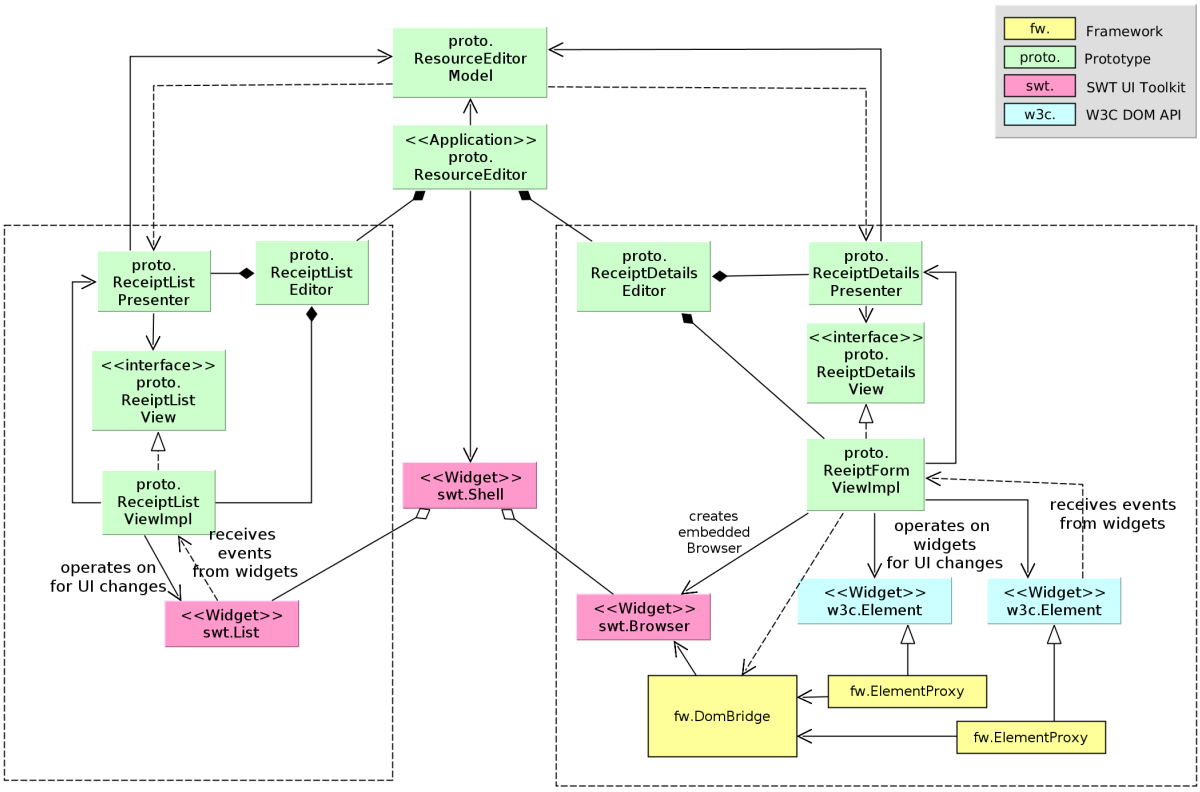


Abbildung 4.6.: Klassendiagramm des Prototyps unter Nutzung des Rahmenwerks

Der `ReceiptEditor` besitzt selbst eine View `ReceiptEditorViewImpl`, die den Oberflächen-Rahmen für die Views der /Unter-Editoren `ReceiptListEditor` und `ReceiptDetailsEditor` vorgibt. Einen Presenter besitzt der `ReceiptEditor` selbst nicht, da keine Präsentationslogik vorliegt. Die beiden /Unter-Editoren `ReceiptListEditor` und `ReceiptDetailsEditor` werden vom `ReceiptEditor` erzeugt, verknüpfen sich mit dem `ResourceEditorModel` und hängen ihre Views in den Oberflächen-Rahmen ein. Damit ist der `ReceiptEditor` bereit und steht dem Benutzer zur Verfügung.

4.2.2. ReceiptListEditor

Wie in Bild 3.3 links zu sehen ist, werden die Belege im /Beleg-Editor als Liste dargestellt. Einer der Belege kann selektiert sein. Zusätzlich steht ein Neu-Button zur Verfügung, über den durch Klick ein neuer Beleg erzeugt und ausgewählt werden kann.

Diese Funktionalität wird durch mehrere Klassen im Zusammenspiel realisiert. Als Ein-

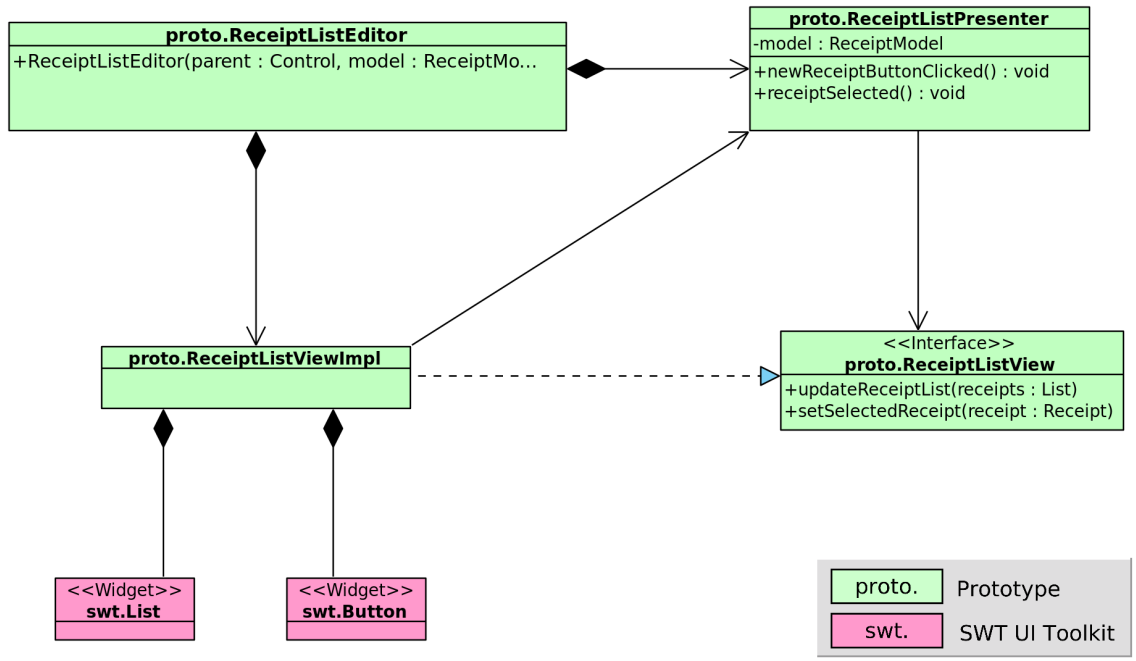


Abbildung 4.7.: Design des ReceiptListEditor

stiegspunkt dient hier der `ReceiptListEditor`. Dessen Aufgabe ist es, die Klassen `ReceiptListPresenter` und `ReceiptListViewImpl` zu instantiieren und miteinander zu verknüpfen:

Der `ReceiptListPresenter` realisiert hier die Präsentationslogik, die die Listendarstellung, die Belegselektion und das Klicken auf den Neu-Button behandelt. Die Darstellung der Belege initiiert der `ReceiptListPresenter` indem er die Belegliste aus dem `ReceiptEditorModel` an die `ReceiptListView` überträgt und diese anweist, die Liste darzustellen. Dies fordert der `ReceiptListPresenter` initial an, wenn er mit der View-Implementierung `ReceiptListViewImpl` verknüpft wird. Der `ReceiptListPresenter` empfängt Ereignisse des `ReceiptEditorModel`, da er sich am Model als Observer registriert hat. So weist er die `ReceiptListViewImpl` auch bei Änderungen in der Belegliste oder an einzelnen Belegen an, die dargestellte Liste zu aktualisieren. Das gleiche gilt für die Belegselektion. Der laut `ReceiptEditorModel` ausgewählte Beleg wird initial und nach Benachrichtigung über die Änderung der Selektion oder der Belegliste auf Anweisung des `ReceiptListPresenter` in der `ReceiptListViewImpl` aktualisiert. Ebenso kann der `ReceiptListPresenter` auch die Benachrichtigung über eine neue Belegselektion von der `ReceiptListView` erhalten. Diese Anforderung setzt der `ReceiptListPresenter` in eine Operation auf

dem `ReceiptEditorModel` um, die dann zur möglicherweise veränderten Belegselektion führt.

Die `ReceiptListViewImpl` kapselt die Operationen mit der Oberflächen-Technologie. Da diese View in der Analyse als auf der Oberflächen-Technologie SWT basierend spezifiziert ist, erstellt sie ihre Widgets mit Hilfe von SWT selbstätig. Diese Widgets sind die List-Box und der Neu-Button. Die List-Box wird in der Instanz der View nach der Erzeugung gespeichert und steht so für die Sondierung und Aktualisierung ihres Inhalts auf Anforderungen durch den `ReceiptListPresenter` zur Verfügung. An den Neu-Button wird ein Listener angehängt, so dass der Klick auf den Button an den `ReceiptListPresenter` bei Auftreten weitergeleitet werden kann.

4.2.3. ReceiptDetailsEditor

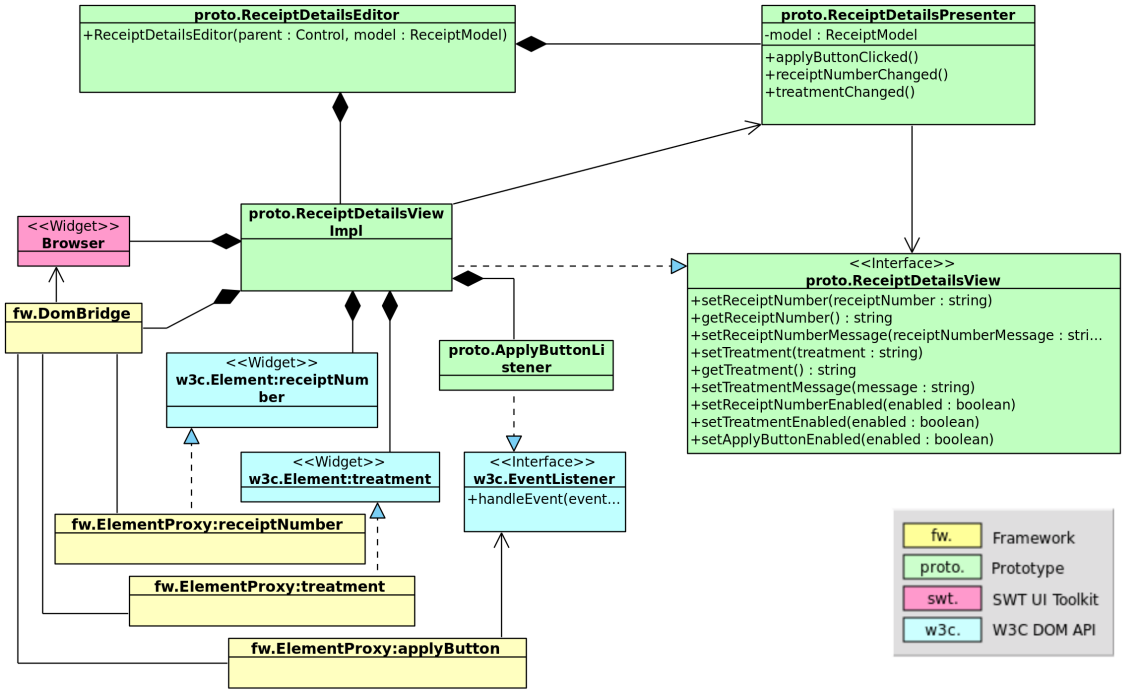


Abbildung 4.8.: Design des ReceiptDetailsEditor

Der `ReceiptDetailsEditor`, in Abbildung 3.3 rechts zu sehen, dient der Darstellung und Bearbeitung der Details eines Belegs, hier durch einfach Textfelder repräsentiert. Der Übernehmen-Button dient der Übernahme von Änderungen an den Details.

Der Aufbau des `ReceiptDetailsEditor` erfolgt analog zum Aufbau des `ReceiptListEditor` nach dem MVP-Muster. Die verschiedenen Kollaborateure, hier

ReceiptDetailsPresenter und ReceiptDetailsViewImpl, werden vom ReceiptDetailsEditor erzeugt und miteinander verbunden.

Der ReceiptDetailsPresenter steuert die Befüllung der Formularfelder und deren Status. Der ReceiptDetailsPresenter überträgt also die Daten des aktuell selektierten Belegs aus dem ReceiptEditorModel in die Textfelder der ReceiptDetailsViewImpl. Dies erfolgt initial nachdem die View Bereitschaft signalisiert und beim Ereignis über die Änderung des aktuell selektieren Belegs, über das der ReceiptDetailsPresenter benachrichtigt wird, da er sich als Observer am ReceiptEditorModel registriert hat. Über Änderungen im Formular wird der ReceiptDetailsPresenter von der ReceiptDetailsViewImpl informiert, so dass er die Textfelder validieren und evt. einen Hinweis für den Benutzer einblenden kann. In diesem Fall muss auch der Übernehmen-Button in einen Modus geschaltet werden, der den Klick darauf verhindert. Wird der Übernehmen-Button vom Benutzer angeklickt, so aktualisiert der ReceiptDetailsPresenter den bearbeiteten Beleg im ReceiptEditorModel aus den aktuellen Daten in den Textfeldern der ReceiptDetailsViewImpl.

Die ReceiptDetailsViewImpl realisiert ihre Oberfläche auf Basis der Web-Oberflächen-Technologien. Hierzu bettet die View einen Browser ein und instantiiert eine DomBridge. Über die DomBridge lädt die View ihre Oberfläche als HTML-Seite in den eingebetteten Browser. Damit die View analog zur ReceiptListView die Oberfläche sondieren und ändern kann, benötigt sie Zugriff auf die dargestellten Widgets. Diese werden im Browser durch DOM-Elemente repräsentiert und stehen durch die DomBridge dem Anwendungsentwickler in der ReceiptDetailsViewImpl zur Verfügung. Nachdem der Ladevorgang der HTML-Seite abgeschlossen ist, lokalisiert die ReceiptDetailsViewImpl unter Benutzung der DomBridge die benötigten Widget-DOM-Elemente. Im vorliegenden Fall sind das die Textfelder und der Übernehmen-Button. Die Lokalisierung kann hier durch Zugriff über die ID, Navigation über den DOM-Baum oder z.B. per XPath erfolgen. Über die DOM-Elemente können die dargestellten Oberflächen-Widgets aus dem Browser nun analog wie im ReceiptListViewImpl beschrieben sondiert und verändert werden. An das Element des Übernehmen-Buttons muss die ReceiptDetailsViewImpl einen EventListener anhängen, um über das Klick-Ereignis benachrichtigt zu werden, so dass der ReceiptDetailsPresenter darüber informiert werden kann. Dies gilt analog für Veränderungen an den Textfelder. Hierfür wird ebenfalls ein EventListener registriert, damit der ReceiptDetailsPresenter informiert werden kann und damit die Möglichkeit zur Validierung der Eingaben erhält.

4.3. Zusammenfassung

In diesem Abschnitt wurde ein Entwurf für das Rahmenwerk erstellt. Die Problemstellungen der Versorgung des Browsers mit benötigten Ressourcen, der Zugriff auf den DOM über Java-Proxy-Objekte und der Empfang von Ereignissen aus dem Browser-DOM wurden gelöst.

Der Entwurf des Prototyps gibt den Nachweis über die Funktionsfähigkeit des Rahmenwerks und zeigt die Anwendung des Model-View-Presenter-Musters wie es als Architekturvorlage in der Analyse beschrieben wurde.

Im folgenden Kapitel wird die Umsetzung des Entwurfs und die aus der Realisierung gewonnenen Erkenntnisse beschrieben. Die einzelnen Komponenten werden anhand von Code-Fragmenten gezeigt und es werden Detail-Fragestellungen gelöst: Kommunikation mit dem nativen Broser-DOM durchgeführt, Erzeugung und Verknüpfung der MVP-Strukture und das Testen der Präsentationslogik.

5. Realisierung

Im Folgenden wird die Realisierung von Rahmenwerk und Prototyp beschrieben. Der Ablauf folgt den im Design entworfenen Komponenten.

Wie im Design angesprochen wird die `ResourceBridge` mit einem eingebetteten HTTP-Server erweitert, um dem Browser die angefragten Ressourcen über eine lokale TCP/IP-Socket zur Verfügung zu stellen. Der Abschnitt über die Realisierung der `DomBridge` erläutert, wie mit dem eingebetteten Browser über JavaScript kommuniziert und die einzelnen DOM-Knoten adressiert werden. Um die aufgestellten Konzepte schnell überprüfen zu können wurde JavaScript für die Anbindung der `DomBridge` an den Browser gewählt. Da dieser Aspekt vor dem Anwendungsentwickler vollständig verborgen ist, kann die JavaScript-Lösung durch eine Anbindung über die native Schnittstelle des Browsers ausgetauscht werden, die vermutlich eine höhere Performance aufweist.

Ferner wird die Realisierung des Prototyps weiter expliziert. Dies gilt insbesondere für den Aspekt wie Model, View und Presenter konstruiert und miteinander verbunden werden können. Dies wird im Abschnitt `ResourceListEditor` beschrieben. Wie mit der asynchronen Initialisierung der Browser-View umgegangen wird, erläutert der Abschnitt 5.2.4 über `ReceiptEditorView` und `ReceiptEditorViewImpl`.

5.1. Rahmenwerk

5.1.1. ResourceBridge

Aufgabe der `ResourceBridge` ist die Versorgung des Browsers mit den Ressourcen, die durch das vom Rahmenwerksnutzer implementierte Verfahren bereitgestellt werden. Wie im Design der `ResourceBridge` (4.2) vorgegeben, erfolgt dies über die `ResourceProvider`-Implementierungen, die an der `ResourceBridge` angemeldet werden müssen. Hierbei werden alle registrierten `ResourceProvider` der Reihe nach durchlaufen bis der erste Provider die `Resource` liefern kann. Die Implementierung des Ablaufs zeigt der folgende Code-Ausschnitt 5.1:

```
1  public Resource findResource(String path) throws IOException {
2
3          for (ResourceProvider curr : resourceProviders) {
4
5                  Resource resourceFound = curr.resolveResource(path);
6                  if (resourceFound != null) {
7                          return resourceFound;
8                  }
9          }
10
11         return null;
12 }
```

Listing 5.1: Auffinden von Resources durch die ResourceBridge mit Hilfe seiner ResourceProvider

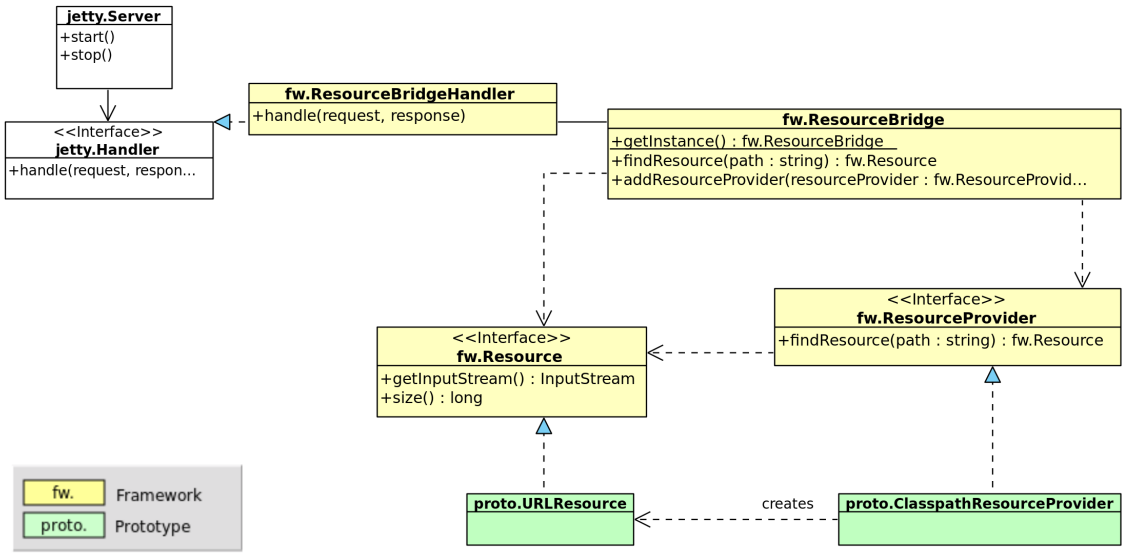


Abbildung 5.1.: Klassendiagramm der ResourceBridge mit embedded HTTP-Server

Das Klassendiagramm (Abbildung 5.1) zeigt zusätzlich zum Design der `ResourceBridge` die Strategie `ClasspathResourceProvider`, die Ressourcen aus dem Klassenpfad bereitstellt. Da Ressourcen aus dem Klassenpfad als URL referenzierbar sind, ist die ausgelieferte Ressource als Wrapper (vgl. Gamma u. a. 1995, S. 139) um das Objekt `java.net.URL` realisiert (Klasse `URLResource` in Abbildung 5.1). Wird der `ClasspathResourceProvider` als Strategie an der `ResourceBridge` registriert, steht der gesamte Klassenpfad der Java-Anwendung als Quelle für Ressourcen zur Verfü-

gung. So können HTML-Seiten, CSS-Dateien und andere Mediendateien behandelt werden wie andere Anwendungs-Ressourcen wie z.B. Properties-Dateien.

Für die Anbindung an den Browser wurde bereits im Design für den Prototypen der Einsatz eines lokalen HTTP-Servers festgelegt. Hierbei findet ein eingebetteter Webserver[1] Anwendung. Dieser wird bei der ersten Nutzung der `ResourceBridge`, also bei der Anmeldung der ersten Strategie, gestartet. Der Request-Handler `ResourceBridgeHandler` verbindet HTTP-Server und `ResourceBridge`.

5.1.2. DOM-Bridge

Die `DomBridge` dient als Einstiegspunkt für den Zugriff auf den nativen Browser-DOM. Sie wird vom Implementierer der View mit dem SWT-Widget `Browser` des eingebetteten Browsers instantiiert.

```
1  public abstract class DomBridge {
2
3          public static interface DocumentReadyCallback {
4                  void ready();
5          }
6
7          public static DomBridge newInstance(Browser browser,
8                          Class<?> viewClass) {
9                  return new DomBridgeImpl(browser, viewClass);
10         }
11
12         public abstract void startLoad(String path,
13                         DocumentReadyCallback documentReadyCallback);
14
15         public abstract Document getDocument();
16 }
```

Listing 5.2: Schnittstelle der DomBridge

Die `DomBridge` stellt Proxies auf die verschiedenen `Node`-Objekte des nativen Browser-DOMs zur Verfügung, wie z.B. das `Document` über die Methode `getDocument()`. Da das W3C-DOM-Modell sehr umfangreich ist, wird das Beispiel exemplarisch auf die `Node`-Typen `Document` und `Element` beschränkt. Mit diesen `Node`-Typen ist eine Anwendung wie der /Beleg-Editor bereits realisierbar. Die `Node`-Objekte der beiden werden in den Klassen `JsDomDocumentProxy` und `JsDomElementProxy` realisiert. (s. Klassendiagramm 5.2).

[1] Quelle: http://jetty.codehaus.org/jetty/

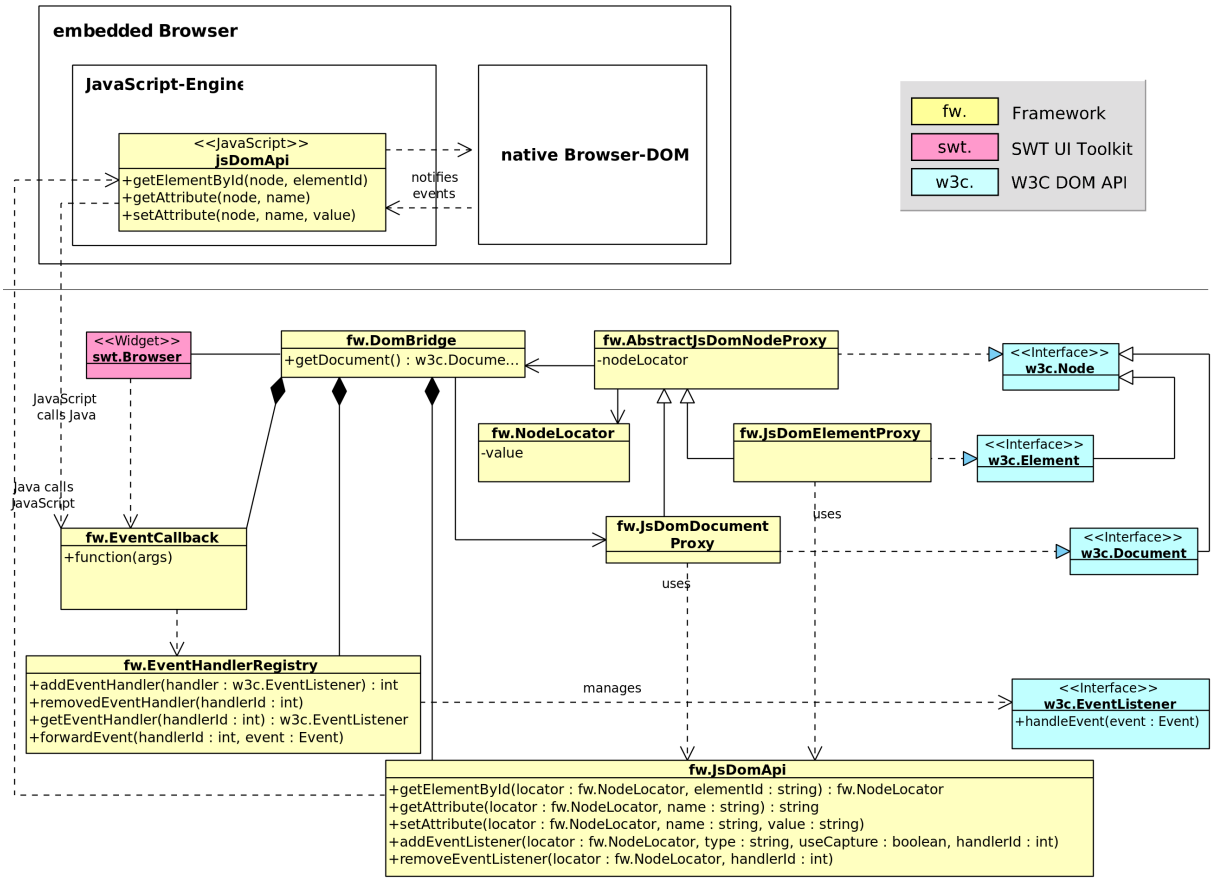


Abbildung 5.2.: Klassendiagramm der DomBridge mit Node-Proxies und EventHandlerRegistry

Um eine DOM-`Node` adressieren zu können, wird ein `NodeLocator` eingeführt. Der `NodeLocator` kann in einen JavaScript-Ausdruck umgewandelt werden, damit die `Node` im nativen DOM-Baum des Browsers leicht aufgelöst werden kann. Beispiele für `NodeLocator`-Ausdrücke zeigt die Tabelle 5.1.

`NodeLocator`-Ausdruck	Typ	JavaScript-Ausdruck
document	literal	document
/ oder /html	XPath	document.documentElement
#example	ID	document.getElementById('example')

Tabelle 5.1.: Beispiele für gültige `NodeLocator`-Ausdrücke

Da alle `Node`-Proxy-Klassen von `AbstractJsDomNodeProxy` erben, besitzen sie alle neben ihrem `NodeLocator` auch eine Referenz auf die Klasse `JsDomApi`. Diese ist eine Sammlung aller Funktionen, die von den Java-DOM-Node-Proxies auf dem nativen Browser-DOM aufgerufen werden können. Wird eine DOM-Operation-Methode auf einem Java-Proxy aufgerufen, ist durch den Aufrufempfänger implizit festgelegt, auf welches Objekt die Funktionalität angewandt werden soll. Dies fehlt allerdings im Übergang zum nativen DOM über die `JsDomApi`, die eine rein funktionale Schnittstelle darstellt. Hierfür muss das aktuelle `'this'` durch einen `NodeLocator` mit an die `jsDomApi`-Funktion übergeben werden. Dies ist jeweils der erste Parameter in einer Funkton der `JsDomApi`. Der `NodeLocator` wird in einen JavaScript-Ausdruck umgewandelt und so an die `jsDomApi`-Funktion übergeben. Der folgende Code-Ausschnitt 5.3 zeigt die Java-Klasse `JsDomApi` mit elementaten Funktionen, die sie für die verschiedenen Proxy-Typen realisiert. Die Entsprechungen im JavaScript-Objekt `jsDomApi` folgen in Listing 5.4:

```
1  public NodeLocator getElementById(NodeLocator node, String id) {
2
3          String scriptTemplate = "return jsDomApi.getElementById(%s, '%s')";
4          String script = String.format(scriptTemplate, node.toJavaScript(), id);
5          JSONObject result = evaluateSafely(script);
6          String locatorStr = (String) result.get("locator");
7          return NodeLocator.valueOf(locatorStr);
8  }
9
10 public String getAttribute(NodeLocator node, String name) {
11
12         String scriptTemplate = "return jsDomApi.getAttribute(%s,'%s')";
13         String script = String
14                         .format(scriptTemplate, node.toJavaScript(), name);
15         JSONObject result = evaluateSafely(script);
16         String value = (String) result.get("value");
17         return value;
```

```
18  }
19
20  public void setAttribute(NodeLocator node, String name, String value) {
21
22          String scriptTemplate = "jsDomApi.setAttribute(%s,'%s','%s')";
23          String script = String
24                  .format(scriptTemplate, node.toJavaScript(), name);
25          evaluateSafely(script, String.class);
26  }
27
28  public void addEventListener(NodeLocator node, String type, int listenerId,
29                  boolean useCapture) {
30          String scriptTemplate = "return jsDomApi.addEventListener(%s,%s,'%s', %s)";
31          String script = String.format(scriptTemplate, node.toJavaScript(),
32                  listenerId, type, useCapture);
33          evaluateSafely(script);
34  }
35
36  private JSONObject evaluateSafely(String js) {
37          try {
38                  String resultStr = (String) browser.evaluate(js);
39                  JSONObject result = (JSONObject) (new JSONParser())
40                          .parse(resultStr);
41                  if (result == null)
42                          throw new RuntimeException("Result was null.");
43                  String status = (String) result.get("status");
44                  if (!"ok".equals(status))
45                          throw new RuntimeException(status);
46                  return result;
47          } catch (ParseException ex) {
48                  throw new RuntimeException(ex);
49          }
50  }
51
52  private void executeSafely(String js) {
53          this.browserWidget.execute(js);
54  }
```

Listing 5.3: Ausschnitt aus JsDomApi

```
1   jsDomApi = new function() {
2
3           var self = this;
4
5           this.getElementById = function(node, id) {
6                   var elementLocator = self.locatorOf(node.getElementById(id));
7                   return JSON.stringify({
8                           locator : elementLocator,
```

```
9                       status : 'ok'
10                  });
11          };
12
13          this.getAttribute = function(node, name) {
14                  var value = node[name];
15                  return JSON.stringify({
16                          value : value,
17                          status : 'ok'
18                  });
19          };
20
21          this.setAttribute = function(node, name, value) {
22                  node[name] = value;
23                  return JSON.stringify({
24                          status : 'ok'
25                  });
26          };
27
28          this.addEventListener = function(node, listenerId, type, capture) {
29                  node.addEventListener(type, function() {
30                          jsDomApiSendEvent(listenerId, null);
31                  }, capture);
32
33                  return JSON.stringify({
34                          status : 'ok'
35                  });
36          };
37
38          ...
39  }
```

Listing 5.4: Ausschnitt aus jsDomApi.js (JavaScript)

Die Implementiertung der `getAttribute()`-Methode `Element`-Proxy Klasse `JsDomElementProxy` mit Hilfe der `JsDomApi` geschieht wie in Listing 5.5 beschrieben:

```
1  @Override
2  public String getAttribute(String name) {
3          return this.domBridge.getJsDomApi().getAttribute(this.locator, name);
4  }
```

Listing 5.5: JsDomElementProxy

Damit an einem DOM-`Node`, der durch einen `AbstractJsDomNodeProxy` repräsentiert wird, ein `EventListener` angemeldet werden kann, müssen die konkreten

JsDomProxy-Unterklassen das Interface org.w3c.events.EventTarget erfüllen. Hier kann über addEventListener() ein EventListener angemeldet werden. Die Listener-Implementierung wird in der EventHandlerRegistry der aktuellen DomBridge abgelegt und erhält eine generierte ID. Über die jsDomApi wird nun im nativen Browser-DOM ein Event-Listener an den jeweiligen Knoten angehängt, der bei Auftreten des jeweiligen Ereignisses einen vom Rahmenwerk bereitgestellten Browser-Callback aufruft, der dieses Event in den Java-Raum weiterleitet. Hierbei wird die ID des empfangenden Event-Listeners mit überliefert. Der Browser-Callback löst nun anhand der Listener-ID mit Hilfe der EventHandlerRegistry die EventListener-Instanz auf und leitet das Event an ihn weiter. Die Registrierung des EventListener erfolgt wie in Listing 5.6 beschrieben:

```
1 public void addEventListener(String type, EventListener listener, boolean useCapture) {
2         int eventListenerId = this.domBridge.getEventListenerRegistry()
3                         .registerEventListener( listener );
4         this.domBridge.getJsDomApi().addEventListener(locator, type,
5                         eventListenerId, useCapture);
6 }
```

Listing 5.6: Ausschnitt aus JsDomNodeProxy

Die Auflösung des EventListeners aus der Listener-ID und die Auslieferung des Ereignisses an den entsprechenden Java-Event-Listener zeigt der Code-Ausschnitt in Listing 5.7, das aus der Klasse DomBridgeImpl stammt.

```
1 new BrowserFunction(browser, "jsDomApiSendEvent") {
2         @Override
3         public Object function(Object[] arguments) {
4                 getEventListenerRegistry().forwardEvent(((Number) arguments[0]).intValue(),
5                         null);
6                 return null;
7         }
8 };
```

Listing 5.7: Ausschnitt aus JsDomNodeProxy

JavaScript und Java besitzen beide Laufzeitumgebungen mit automatischer Speicherbereinigung (Garbage Collection). In beiden System werden Objekte, die nicht mehr benötigt werden, weil kein anderes Objekt mehr eine Referenz darauf hält, automatisch aus dem Speicher entfernt. Wird nun ein Element aus dem nativen DOM-Baum entfernt, so wird der Event-Listener auf JavaScript-Seite nicht mehr referenziert und daher aus dem Speicher entfernt. Die Referenz des Java-seitigen EventListeners befindet sich jedoch noch in der EventHandlerRegistry und es gibt auch Browser-seitig keine Möglichkeit über das Entfernen eines Objekts informiert zu werden. Eine Lösung für dieses Problem ist z.B. das

Abfangen der Mitteilung über die Löschung von DOM-Knoten und dem anschließenden Entfernen aller betroffenen Event-Listenern aus dem Java-Raum. Da eine Lösung zur Verfügung steht und der Aspekt auch für Überprüfung der aufgestellten Konzepte nur nebensächlich ist, deckt der Prototyp diese Funktionaliät auch nicht ab.

5.2. Prototyp

5.2.1. ReceiptListEditor

Die Klasse `ReceiptListEditor` repräsentiert den linksseitigen /Unter-Editor des /Beleg-Editors. Aus technischer Sicht dient die Klasse als Einstieg für den übergeordneten Editor. Ihre Aufgabe ist es, den `ReceiptListPresenter` und die `ReceiptListViewImpl` zu instantiieren und miteinander zu verknüpfen. Der `ReceiptListEditor` enthält Wissen über die verwendete Oberflächen-Technologie und die tatsächliche View-Implementierung, die dem `ReceiptListPresenter` nur über die abstrakte View-Schnittstelle bekannt ist. Dieses Wissen über verschiedene Verantwortlichkeiten stellt in diesem Zusammenhang kein Problem dar, da die Klasse sehr übersichtlich ist und nur dem Zweck der Konstruktion dient.

```
1 public ReceiptListEditor(Composite parent, int style, ReceiptEditorModel model) {
2
3         ReceiptListPresenter listPresenter = new ReceiptListPresenter(model);
4         ReceiptListViewImpl listView = new ReceiptListViewImpl(listPresenter);
5         listView.init(parent, SWT.NONE);
6         listPresenter.setView(listView);
7 }
```

Listing 5.8: Erzeugung von ReceiptListPresenter und ReceiptListViewImpl im Konstruktor des ReceiptListEditor

5.2.2. ReceiptListPresenter

Der `ReceiptListPresenter` implementiert die Steuerung der `ReceiptListView`. Seine Aufgabe ist es auf `ReceiptModel`-Änderungen zu reagieren (vgl. hierzu `receiptSelected()` und `receiptsChanged()` in den Zeilen 3-13) und diese in der `ReceiptListView` sichtbar zu machen (Methode `updateView()`) und Selektionsänderungen durch den Benutzer, die von der `ReceiptListViewImpl` an ihn übermittelt werden (Methode `receiptSelected()`), auf dem `ReceiptModel` durchzuführen, s. hierzu die Zeilen 21 und 22.

```
1  public ReceiptListPresenter(ReceiptEditorModel model) {
2          this.model = model;
3          this.model.addListener(new ReceiptEditorModel.Listener() {
4                  @Override
5                  public void receiptSelected() {
6                          updateView();
7                  }
8
9                  @Override
10                 public void receiptsChanged() {
11                         updateView();
12                 }
13         });
14 }
15
16 public void newReceiptButtonClicked() {
17         this.model.createAndSelectReceipt();
18 }
19
20 public void receiptSelected(int idx) {
21         Receipt receipt = this.model.getReceiptList().get(idx);
22         this.model.selectReceipt(receipt, true);
23 }
24
25 private void updateView() {
26         this.view.updateReceiptList(
27                         ReceiptListPresenter.this.model.getReceiptList(),
28                         ReceiptListPresenter.this.model.getSelectedReceipt());
29 }
```

Listing 5.9: Registrierung des ReceiptListPresenter am ReceiptEditorModel

5.2.3. ReceiptListView und ReceiptListViewImpl

Die Schnittstellte der `ReceiptListView` sieht das Aktualisieren der Belegliste vor. Diese Methode wird vom `ReceiptListPresenter` aufgerufen.

```
1  public interface ReceiptListView extends View {
2          void updateReceiptList(List<Receipt> receipts, Receipt selectedReceipt);
3  }
```

Listing 5.10: Interface der ReceiptListView

Die Implementierung dazu, die Klasse `ReceiptListViewImpl`, realisiert ihre Oberfläche durch das instantiieren von SWT-Widgets und ist in diesem Zusammenhang nicht rele-

vant, da die Oberflächen-Entwicklung auf der Basis von Web-Oberflächen-Technologien im Zentrum der Betrachtung steht. Der Quellcode ist zum Vergleich im Anhang A.1 hinterlegt.

5.2.4. ReceiptDetailsView und ReceiptDetailsViewImpl

Eine auf Web-Oberflächen-Technologien basierende View-Implementierung besteht aus einer Java-View-Implementierung und aus einer Oberflächen-Beschreibung in Form einer HTML-Seite und optional weiteren Ressourcen.

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4          <link rel="stylesheet" type="text/css" href="view.css"></link>
5  </head>
6  <body>
7          <fieldset>
8                  <legend>Receipt Details</legend>
9                  <div class="attribute">
10                         <label for="receiptNumber">Receipt Number:</label>
11                         <div class="formField">
12                                 <input type="text" id="receiptNumber" disabled="disabled" />
13                                 <div id="receiptNumberMessage" class="message"></div>
14                         </div>
15                 </div>
16                 <div class="attribute">
17                         <label for="treatment">Treatment:</label>
18                         <div class="formField">
19                                 <textarea id="treatment" disabled="disabled"></textarea>
20                                 <div id="treatmentMessage" class="message"></div>
21                         </div>
22                 </div>
23                 <button id="applyButton">Apply</button>
24         </fieldset>
25 </body>
26 </html>
```

Listing 5.11: HTML-Seite der ReceiptDetailsViewImpl

Die Schnittstelle der `ReceiptDetailsView` ist wie folgt definiert. Neben dem Setzen und Lesen von Feldinhalten sind hier auch Funktionen zur Steuerung des Oberflächen-Zustands enthalten. Dies ist explizit gewollt, da diese Funktionalität Teil der Präsentationslogik ist und damit in der Verantwortung des Presenters liegt.

```
1  public interface ReceiptDetailsView extends View {
2
```

```
3           void setReceiptNumber(String receiptNumber);
4
5           void setReceiptNumberMessage(String receiptNumberMessage);
6
7           String  getReceiptNumber();
8
9           void setTreatment(String treatment);
10
11          void setTreatmentMessage(String treatmentMessage);
12
13          String  getTreatment();
14
15          void setApplyEnabled(boolean enabled);
16
17          void setReceiptNumberEnabled(boolean hasReceipt);
18
19          void setTreatmentEnabled(boolean hasReceipt);
20  }
```

Listing 5.12: Interface der ReceiptDetailsView

Wenn die `ReceiptDetailsViewImpl` über `init()` (s. Listing 5.13, Zeile 5) aufgefordert wird ihre Oberfläche zu erstellen, muss sie die `DomBridge` instantiieren und sie anweisen, die zur View gehörige HTML-Seite zu laden. Dies erfolgt durch den Aufruf von `startLoad()`.

Im nächsten Schritt muss die `ReceiptDetailsViewImpl` die für die weitere Arbeit benötigten DOM-`Nodes` lokalisieren. Da der Browser seinen Inhalt asynchron lädt, kann man nicht nach der Rückkehr von `startLoad()` annehmen, dass die HTML-Seite mit abhängigen Ressourcen vollständig geladen ist. Hierzu dient der `DocumentReadyCallback`, dessen `ready()`-Methode aufgerufen wird, sofern der eingebettete Browser die Seite geladen und die Oberfläche dargestellt hat und der DOM vollständig zur Verfügung steht.

```
1   @Override
2   public Control  init (Composite parent, int style) {
3           this.browser = new Browser(parent, SWT.WEBKIT);
4           this.domBridge = new DomBridge(this.browser, getClass());
5           this.domBridge.startLoad("view.html",
6                           new DomBridge.DocumentReadyCallback() {
7                                   @Override
8                           public void ready() {
9                                   initWidgets ();
10                          }
11                  });
12          return browser;
13  }
```

Listing 5.13: Initialisierung der ReceiptDetailsViewImpl

Steht der DOM vollständig zur Verfügung, kann die `ReceiptDetailsViewImpl` nun die benötigten DOM-`Nodes` bzw. Proxies lokalisieren und sich Referenzen für den weiteren Zugriff darauf speichern. Im folgenden Code-Abschnitt wird exemplarisch gezeigt, wie die DOM-Elemente der Textfelder `receiptNumber` und `treatment` lokalisiert und für die weitere Arbeit gespeichert werden. Das DOM-`Element` für den Übernehmen-Button wird lokalisiert und zur Anmeldung eines `EventListeners` genutzt, so dass bei Klicken auf diesen Button, die Information darüber an den `ReceiptDetailsPresenter` weiter gereicht wird (`applyClicked()` in Listing 5.14, Zeile 12).

```
1  private void initWidgets() {
2          Document doc = this.domBridge.getDocument();
3          this.receiptNumberText = doc.getElementById("receiptNumber");
4          this.treatmentText = doc.getElementById("treatment");
5
6          Element applyButton = doc.getElementById("applyButton");
7          ((EventTarget) applyButton).addEventListener("click",
8                  new EventListener() {
9
10                         @Override
11                         public void handleEvent(Event evt) {
12                                 presenter.applyClicked();
13                         }
14                 }, true);
15 }
```

Listing 5.14: Initialisierung der Widgets in ReceiptDetailsViewImpl mit EventListener

In der Method `setReceiptNumberEnabled()` kann man gut eine Schwäche der Entscheidung für die W3C-DOM-API sehen. Die Attribute von Elementen sind in der API alle `String`-basiert. Eigentlich handelt es sich bei dem `Attribute disabled` in Listing 5.15, Zeile 15 und 16, um ein `Attribute` des Typs `boolean`, so dass dies explizit konvertiert werden muss.

```
1  @Override
2  public void setReceiptNumber(String value) {
3
4          this.receiptNumberText.setAttribute("value", value);
5  }
6
7  @Override
8  public String getReceiptNumber() {
9
10         return this.receiptNumberText.getAttribute("value");
11 }
```

```
12
13 @Override
14 public void setReceiptNumberEnabled(boolean enabled) {
15         this.receiptNumberText.setAttribute("disabled",
16                         String.valueOf(!enabled));
17 }
```

Listing 5.15: Setzen und Auslesen der receiptNumber-Felds in ReceiptDetailsViewImpl

5.2.5. ReceiptDetailsPresenter

Der `ReceiptDetailsPresenter` vermittelt zwischen der `ReceiptDetailsView` und dem `ReceiptModel`. Bei seiner Erstellung meldet sich der Presenter am `ReceiptModel` an, um über Änderungen informiert zu werden.

```
1  public ReceiptDetailsPresenter(ReceiptEditorModel model) {
2          this.model = model;
3          this.model.addListener(new ReceiptEditorModel.Listener() {
4
5                  @Override
6                  public void receiptSelected() {
7                          onReceiptSelected();
8                  }
9
10                 @Override
11                 public void receiptsChanged() {
12                         onReceiptSelected();
13                 }
14         });
15 }
```

Listing 5.16: Auszug aus ReceiptDetailsPresenter

Der folgende Code-Abschnitt demonstriert, dass die Validierung von Eingaben frei von Oberflächen-Technologien ist, was ein wesentliches Kriterium für die Anwendung des Model-View-Presenter-Musters war.

```
1  public void receiptNumberChanged() {
2          validate ();
3  }
4
5  public void treatmentChanged() {
6          validate ();
7  }
8
```

```
 9  private void validate () {
10          boolean hasReceipt = this.model.hasSelectedReceipt();
11          if (!hasReceipt) {
12                  this.view.setReceiptNumberEnabled(false);
13                  this.view.setTreatmentEnabled(false);
14                  this.view.setApplyEnabled(false);
15                  return;
16          }
17
18          this.view.setReceiptNumberEnabled(true);
19          this.view.setTreatmentEnabled(true);
20
21          String receiptNumber = this.view.getReceiptNumber();
22          boolean receiptNumberValid = !isEmpty(receiptNumber);
23          if (receiptNumberValid)
24                  this.view.setReceiptNumberMessage("");
25          else
26                  this.view.setReceiptNumberMessage("invalid!");
27
28          String treatment = this.view.getTreatment();
29          boolean treatmentValid = !isEmpty(treatment);
30          if (treatmentValid)
31                  this.view.setTreatmentMessage("");
32          else
33                  this.view.setTreatmentMessage("invalid!");
34
35          this.view.setApplyEnabled(receiptNumberValid && treatmentValid);
36  }
```

Listing 5.17: Auszug aus ReceiptDetailsPresenter

Bei Anklicken des Übernehmen-Buttons im `ReceiptDetailsEditor` wird das `ReceiptModel` aufgefordert, die Daten aus den Feldern der `ReceiptDetailsView-Impl` in den selektierten Beleg zu übernehmen. Daraufhin wird das `ReceiptModel` den Beleg ändern und ein Ereignis darüber an alle Observer senden (zu dem Zeitpunkt sind dort `ReceiptListPresenter` und `ReceiptDetailsPresenter` angemeldet), die dann mit der Aktualisierung ihrer Liste respektive ihrer Textfelder reagieren.

```
1  public void applyClicked() {
2
3          int receiptId = this.view.getReceiptId();
4          String newReceiptNumber = this.view.getReceiptNumber();
5          String newTreatment = this.view.getTreatment();
6          this.model.updateReceipt(receiptId, newReceiptNumber,
7                          newTreatment);
8  }
```

Listing 5.18: Auszug aus ReceiptDetailsPresenter

Das folgende Code-Fragment zeigt zwei Unit-Tests[2], die die korrekte Implementierung des `ReceiptDetailPresenter` überprüfen. Diese Tests sind gute Beispiele dafür, wie einfach Präsentationslogik testbar sein kann, wenn sie frei von der Oberflächen-Technologie gehalten wird. Ein solcher Test ist wesentlich schneller herzustellen, durchzuführen und abzuändern als z.B. der Test über einen GUI-Test-Robot oder gar ein Test von Hand.

Die Tests sind beide nach dem gleichen Muster aufgebaut. `model`, `receipt` und `view` werden durch Mock-Objekte[3] ersetzt.

Der `presenter` ist das zu testende Objekt. Der Test `thatModelIsUpdatedOnApplyClicked` verifiziert, dass die Model-Daten aktualisiert werden, wenn der Presenter über das Ereignis `applyClicked()` informiert wird:

Im Block, der als 'given' gekennzeichnet ist, werden die Vorbedindungen des Tests erzeugt (Zeilen 15-18). In `thatModelIsUpdatedOnApplyClicked()` sind dies die Daten, die als Inhalt der Widgets in der View angenommen werden. Im folgenden wird dann in Zeile 20 das Ereignis `applyClicked()` auf dem Presenter ausgelöst. Zeile 23 im mit 'then' bezeichneten Block prüft die Nachbedingungen, die gelten müssen, damit die Funktionsanforderungen an den Presenter bei dem Ereignis `applyClicked()` als erfüllt gelten können. Dies ist in diesem Fall der Übertrag der Daten aus den Vorbedingungen in das Model. Damit ist der Test abgeschlossen.

`thatViewIsUpdatedOnReceiptSelectedEvent` ist analog aufgebaut und überprüft die Aktualisierung der View aus dem Model heraus bei Eintreten des Ereignisses `receiptSelected()`.

```
1  public class ReceiptDetailsPresenterTest {
2  private @Mock ReceiptEditorModel model;
3  private @Mock Receipt receipt;
4  private @Mock ReceiptDetailsView view;
5
6  private ReceiptDetailsPresenter presenter;
7
8  private void setUpPresenter() {
9          presenter = new ReceiptDetailsPresenter(model);
10         presenter.setView(view);
11 }
12
13 @Test public void thatModelIsUpdatedOnApplyClicked() throws Exception {
```

[2] basierend auf JUnit (junit) und Mockito (mockito)

[3] Mock-Objekte werden eingesetzt, um Abhängigkeiten von zu testenden Klassen durch Stellvertreter zu ersetzen. Die Stellvertreter können, ohne wiederum die Abhängigkeiten der Abhängigkeiten berücksichtigen zu müssen, mit dem erforderlichen Verhalten ausgestattet werden.

```
14          // given
15          setUpPresenter();
16          when(view.getReceiptId()).thenReturn(2);
17          when(view.getReceiptNumber()).thenReturn("R2");
18          when(view.getTreatment()).thenReturn("treatment");
19
20          // when
21          presenter.applyClicked();
22
23          // then
24          verify (model).updateReceipt(2, "R2", "treatment");
25  }
26  @Test public void thatViewIsUpdatedOnReceiptSelectedEvent() throws Exception {
27          // given
28          setUpPresenter();
29          when(model.getSelectedReceipt()).thenReturn(receipt);
30          when(model.hasSelectedReceipt()).thenReturn(true);
31          when(receipt.getReceiptId()).thenReturn(2);
32          when(receipt.getReceiptNumber()).thenReturn("R1");
33          when(receipt.getTreatment()).thenReturn("no treatment");
34
35          // when
36          presenter.receiptSelected();
37
38          // then
39          verify (model).getSelectedReceipt();
40          verify (view).setReceiptId(2);
41          verify (view).setReceiptNumber("R1");
42          verify (view).setTreatment("no treatment");
43          verify (view).setReceiptNumberEnabled(true);
44          verify (view).setTreatmentEnabled(true);
45  }}
```

Listing 5.19: Unit-Test des ReceiptDetailsPresenter

Die gute Testbarkeit der Präsentationslogik wurde erreicht durch die Trennung von der Oberflächen-Technologie und dem Entwurf klar definierter Schnittstellen gegenüber der View und dem Model. Diese Tests belegen die Funktionsfähigkeit der Präsentationslogik im `ReceiptDetailsPresenter` und stellen bei Ausführung nach Änderungen am Code eine Absicherung gegenüber Regressionen dar.

Damit ist die Entwicklung des Prototypen unter Nutzung des Rahmenwerks in sich vollständig und abgeschlossen.

5.3. Zusammenfassung

In diesem Abschnitt wurde die Realisierung des Rahmenwerks und des Prototyps beschrieben.

Im Rahmenwerk wurden die Komponenten `ResourceBridge` und `DomBridge` realisiert.

- Für die `ResourceBridge`, die die Versorgung des Browsers mit Ressourcen wie HTML-, CSS- und Mediendateien übernimmt, wurde ein eingebetteter HTTP-Server genutzt, um sie an den Browser anzubinden. Dass ein HTTP-Server für die Anbindung an den Browser genutzt worden ist, ist für den Rahmenwerknutzer transparent. D.h. der HTTP-Server kann ohne weiteres durch eine Anbindung an die native WebKit ResourceIO-Schnittstelle ausgetauscht werden.
- Die `DomBridge` wurde auf Basis von JavaScript implementiert und stellt der View-Implementierung das native DOM des Browsers über Java-Proxies zur Verfügung und ermöglicht den Empfang von Ereignissen aus dem Browser-DOM über Java-Listener. Hierdurch kann die View-Implementierung den Browser wie ein herkömmliches Oberflächen-Toolkit nutzen. JavaScript wurde für die Anbindung der DOM-Bridge an den Browser gewählt, um die aufgestellten Konzepte schnell überprüfen zu können. Da die Realisierung der Kommunikation mit dem Browser durch die DOM-Proxies verborgen ist, kann dieser Mechanismus durch eine leistungsfähigere Anbindung über eine native Browser-Schnittstelle ersetzt werden.

Die Implementierung des Prototyps erfolgte wie im Design vorgegeben. Es wurden der Einstiegspunkt `ReceiptEditor` mit `ReceiptEditorModel` und im Anschluss die zwei /Unter-Editoren `ReceiptListEditor` und `ReceiptDetailsEditor` implementiert.

Bei der Realisierung des `ReceiptDetailsEditors` fiel auf, dass der eingebettete Browser seine Oberfläche asynchron lädt. Dies ist eine Abweichung zur rein auf SWT basierenden Vorgehensweise, wie sie im `ReceiptListEditor` angewandt wurde, bei der die View gleich nach dem Aufbau der Oberfläche initial befüllt werden konnte. Gelöst wurde dies durch die verzögerte Initialisierung der View, die auf die Meldung des Browsers über Fertigstellung des Ladevorgangs hin erfolgte. Was im ersten Moment als nachteilig erscheint, nämlich der kompliziertere asynchrone Oberflächen-Aufbau, kann jedoch eine erhebliche Verkürzung der Dauer zum Oberflächen-Aufbau und damit eine Beschleunigung des Startvorgangs von Desktop-Applikationen bringen. Welche Ersparnis dadurch erreicht wird, müsste an anderer Stelle genauer untersucht werden.

Die Anwendung des Model-View-Presenter-Musters hat sich als sehr vielversprechend herausgestellt. Es sind zwar initial vom Programmierer etliche Programmzeilen zu schreiben,

dies resultiert aber - wie die Presenter-Klassen des Prototyps zeigen - in einer Präsentationslogik, die frei von Oberflächen-Technologie und damit sehr gut nachvollziehbar ist. Aufgrund der strengen Trennung von View-Implementierung und Model über Schnittstellen ist der Code gut testbar und dies resultiert in einer verbesserten Wartbarkeit.

Der Einsatz von HTML und CSS für die Realisierung der Desktop-Oberfläche bietet gegenüber der rein manuellen Programmierung der Oberfläche auf Basis des Oberflächen-Toolkits erhebliche Produktivitätsvorteile, da viel handgeschriebener Code entfällt (vgl. hierzu `ReceiptListViewImpl` im Anhang A.1). Gegenüber der Erstellung der Oberfläche auf Basis einer proprietären DSL liegt der Vorteil darin, dass nun ein Web-Oberflächen-Entwickler mit seinen Vorkenntnissen in HTML in die Lage versetzt wird die Oberfläche von Desktop-Anwendungen zu gestalten. Die fortschreitende Weiterentwicklung der Browser, HTML und CSS ist ebenfalls etwas, wovon ein positiver Nutzen erwartet werden kann.

Alles in allem verlief die Realisierung damit, bis auf die unerwartete asynchrone Initialisierung der Oberfläche, wie im Entwurf geplant und das Ergebnis stellt eine gute Ausgangsbasis dar, um die Oberfläche der vorliegenden Desktop-Anwendung gemäß auf der beschriebenen Vorgehensweise zu entwickeln.

6. Schluss

6.1. Zusammenfassung

Die grundlegende Motivation für diese Arbeit ist die Modernisierung der Oberfläche einer vorliegenden Desktop-Anwendung von einer Oberflächen-Technologie (Standard Widget Toolkit) auf die vielversprechenderen Web-Oberflächen-Technologien HTML und CSS. Durch die Nutzung von HTML und CSS soll von der schnell fortschreitenden Entwicklung der Browser und Web-Oberflächen-Technologien auch im Bereich der Desktop-Oberflächen profitiert werden. Da ein großer Aufwand bei der Erstellung von Oberflächen auf die Präsentationslogik entfällt, soll die in bereits Java vorliegende Präsentationslogik weiterhin genutzt werden.

Bei der Suche nach verwandten Arbeiten und Technologien fanden sich Arbeiten, die sich mit der Realisierung von Anwendungsoberflächen unter Nutzung des Browsers außerhalb des Web befassten. Manche nutzten den Browser als Laufzeitumgebung, was erhebliche Einschränkungen in den Freiheitsgraden zur Realisierung von Anwendungen zur Folge hat. Andere Technologien stellten den Browser als einbettbare Komponente zur Verfügung. Keine der Arbeiten hat die Nutzung der Web-Oberflächen-Technologien für die Entwicklung von Desktop-Anwendungen mit direktem Zugriff auf den Browser-DOM aus der Präsentationslogik heraus behandelt. Letztlich hat sich das Standard Widget Toolkit aufgrund seines Reifegrads, der Festlegung auf einen einzigen Browsertyp, eine Browserversion und der einfachen Einbindbarkeit in das vorliegende Softwaresystem als die am vielversprechendste Technologie herausgestellt.

In der Analyse wurde die Trennung von Präsentationslogik und Oberflächen-Technologie untersucht. Dabei wurde das Model-View-Presenter-Muster als Architekturmuster vorgelegt, um die Trennung der Präsentationslogik von der Oberflächen-Technologie zu erreichen und unter Beibehaltung der existierenden Präsentationslogik die Oberflächen-Technologie austauschen zu können. Die funktionalen Anforderungen an das Rahmenwerk konnten auf zwei Kernkomponenten zurückgeführt werden:

- Eine Ressource-Bridge versorgt den Browser mit den Ressourcen HTML-Dateien, CSS-Dateien und z.B. Bilder, die die Anwendung bzw. der Browser zur Laufzeit benötigt.

- Die Komponente DOM-Bridge dient als Verbindung zum Browser aus der Java-Anwendung heraus. Sie soll die bidirektionale Kommunikation mit dem nativen Browser-DOM ermöglichen, womit dieser wie ein herkömmliches Oberflächen-Toolkit zur Darstellung der Oberfläche und zum Empfang der Benutzereingaben benutzt werden kann.

Im Kapitel Design wurden Entwürfe für die Komponenten `ResourceBridge` und `DomBridge` entwickelt. Um das Funktionieren des Rahmenwerks nachzuweisen, liefert das Design einen Entwurf für einen Prototypen. Der Prototyp zeigt die Nutzung des Model-View-Presenter-Musters und nutzt das Rahmenwerk für die Darstellung seiner Oberfläche, womit nachgewiesen wird, dass die Anwendungsentwicklung so durchgeführt werden kann.

Im Abschnitt Realisierung wurden die Entwürfe für `ResourceBridge`, `DomBridge` und den Prototypen erfolgreich umgesetzt. Die Re`sourceBridge` wurde durch einen eingebetteten HTTP-Server an den Browser angebunden. Die `DomBridge` wurde mit Hilfe von JavaScript-Funktions-Aufrufen an das native Browser-DOM angebunden. Dieselbe Vorgehensweise konnte für den Empfang von Ereignissen aus dem Browser-DOM angewandt werden.

Die Realisierung des Prototyps zeigt die erfolgreiche Anwendung des Model-View-Presenter-Musters aus der Architekturvorlage, beschreibt anhand von Code-Ausschnitten die Trennung der Präsentationslogik von der Oberflächen-Technologie und belegt damit die gute Testbarkeit mit Unit-Tests.

Der Abschnitt Realisierung schließt mit der Bewertung des Aufwands für die Entwicklung von Oberflächen unter Nutzung der Web-Oberflächen-Technologien HTML und CSS und fasst den Aufwand für die Anwendung der Architekturvorlage zusammen.

Alles in allem verlief die Realisierung somit nahezu wie im Entwurf geplant und das Ergebnis stellt eine gute Ausgangsbasis dar, um die Oberfläche der vorliegenden Desktop-Anwendung basierend auf der beschriebenen Vorgehensweise zu entwickeln.

6.2. Fazit

Das Rahmenwerk macht die Erstellung von Teil- oder Gesamtoberflächen mit Web-Oberflächen-Technologien in Desktop-Anwendungen unter Implementierung der Präsentationslogik in Java möglich. Es ist in dem Punkt erweiterbar, dass der Rahmenwerk-Nutzer entscheidet, wie und woher die vom Browser verwendeten Ressourcen zur Verfügung gestellt werden. Der Anwendungs-Entwickler nutzt bei der Oberflächen-Programmierung mit

der DOM-Bridge einzig[1] die W3C-spezifizierten Schnittstellen, so dass eine Erweiterung des Rahmenwerks oder der Austausch durch eine andere Implementierung für ihn transparent ist.

Der Prototyp zur Evaluation zeigt, dass die Nutzung von Web-Oberflächen-Technologien für Desktop-Anwendungen möglich und sinnvoll ist. Ebenso zeigt er eine Architekturvorlage basierend auf dem MVP-Muster und wie mit dem Rahmenwerk und der DOM-Bridge Oberflächen erstellt werden können. Der Prototyp weist die Lauffähigkeit auf den Betriebssystem-Plattformen Windows, Mac und Linux nach. Damit wird die Anforderung an Portabilität erfüllt.[2]

6.3. Ausblick

Da die Evaluation positiv ausgefallen ist, steht der Nutzung des Rahmenwerks für die Migration der Anwendungs-Oberfläche des vorliegenden Softwaresystems nichts im Wege.

In weiteren Überlegungen sollte geprüft werden, ob der eingebettete HTTP-Server durch eine Anbindung an WebKits WebResourceIO-Schnittstelle ersetzt werden soll. Die durch den HTTP-Server genutzte TCP/IP-Socket auf einem festgelegten Port ist eine mögliche Kollisionsquelle mit anderen Anwendungen auf einer Maschine und deshalb sollte der Verzicht darauf in Erwägung gezogen werden.

Da die Arbeit des Anwendungs-Entwicklers mit der DOM-Bridge noch einiges an manueller Programmierung erfordert, sollten hier noch andere Zugriffsformen auf den Browser-DOM geprüft werden. Z.B. bietet sich die W3C-Selectors-API (w3c-selectors-api2) an. Es ist auch sinnvoll einen Ansatz zur Komponentenbildung zu untersuchen, um durch die Wiederverwendung von Modulen den Entwicklungsaufwand senken zu können. Dem Entwickler kann durch Dependency Injection der DOM-Elemente das Schreiben von immer wieder auftretendem Code zur Lokation von DOM-Knoten oder die Anmeldung von Event-Listenern erspart werden. Auf jeden Fall sind Web-Oberflächen-Technologien auch für Desktop-Anwendungen ein Thema mit Potential.

[1] bis auf die Instanziierung der DomBridge selbst

[2] Dies ist allerdings weitgehend der Tatsache geschuldet, dass sowohl Java als auch SWT auf diesen Plattform lauffähig sind.

Literaturverzeichnis

[mono] *Project Mono*. – URL http://www.mono-project.com. – abgerufen am 9.8.2012

[Adobe Livedocs ProgrammingHTMLAndJS] *Accessing DOM and JavaScript objects from ActionScript*. – URL http://livedocs.adobe.com/flex/3/html/help.html?content=ProgrammingHTMLAndJavaScript_07.html. – abgerufen am 24.5.2012

[Anttonen u. a. 2011] ANTTONEN, Matti ; SALMINEN, Arto ; MIKKONEN, Tommi ; TAIVALSAARI, Antero: Transforming the web into a real application platform: new technologies, emerging trends and missing pieces. In: *Proceedings of the 2011 ACM Symposium on Applied Computing*. New York, NY, USA : ACM, 2011 (SAC '11), S. 800–807. – URL http://doi.acm.org/10.1145/1982185.1982357. – ISBN 978-1-4503-0113-8

[Balzert 1999] BALZERT, Heide: *Lehrbuch der Objektmodellierung. Analyse und Entwurf.* Spektrum Akademischer Verlag, 1999

[Bardin u. a. 2011] BARDIN, Jonathan M. ; LALANDA, Philippe ; ESCOFFIER, Clément ; MURPHY, Alice: Improving user experience by infusing web technologies into desktops. In: *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*. New York, NY, USA : ACM, 2011 (SPLASH '11), S. 225–236. – URL http://doi.acm.org/10.1145/2048147.2048219. – ISBN 978-1-4503-0942-4

[Chang u. a. 2005] CHANG, George ; HSIEH, Jung-Wei ; CALIXTO, Pedro: Components for building desktop-application-like interface in web applications. In: *Proceedings of the 7th Asia-Pacific web conference on Web Technologies Research and Development*. Berlin, Heidelberg : Springer-Verlag, 2005 (APWeb'05), S. 960–971. – URL http://dx.doi.org/10.1007/978-3-540-31849-1_92. – ISBN 3-540-25207-X, 978-3-540-25207-8

[Chirita u. a. 2006] CHIRITA, Paul A. ; FIRAN, Claudiu S. ; NEJDL, Wolfgang: Pushing task relevant web links down to the desktop. In: *Proceedings of the 8th annual ACM international workshop on Web information and data management*. New York, NY, USA : ACM,

2006 (WIDM '06), S. 59–66. – URL http://doi.acm.org/10.1145/1183550.1183563. – ISBN 1-59593-525-8

[Craig Larman 2000] CRAIG LARMAN, Rhett G. (Hrsg.): *Java2 Performance and Idiom Guide*. Prentice Hall PTR, 2000

[Ezra 2007] EZRA, Aviad: *Twisting the MVC Triad - Model View Presenter (MVP) Design Pattern*. Juli 2007. – URL http://aviadezra.blogspot.de/2007/07/twisting-mvp-triad-say-hello-to-mvpc.html

[Ezra 2008] EZRA, Aviad: *MVC (Model View Controller) Design Pattern*. Juni 2008. – URL http://aviadezra.blogspot.de/2008/06/mvc-model-view-controller-design.html

[Fowler 2006a] FOWLER, Martin: *Passive View*. Juli 2006. – URL http://martinfowler.com/eaaDev/PassiveScreen.html

[Fowler 2006b] FOWLER, Martin: *Supervising Controller*. Juni 2006. – URL http://martinfowler.com/eaaDev/SupervisingPresenter.html

[Fowler 2004] FOWLER, Marting: *SoftwareDevelopmentAttitude*. 2004. – URL http://martinfowler.com/bliki/SoftwareDevelopmentAttitude.html. – abgerufen am 13.7.2012

[Gamma u. a. 1995] GAMMA ; HELM ; JOHNSON ; VLISSIDES: *Design Patterns*. Addison-Wesley, 1995

[Gerner u. a. 2006] GERNER, Nicholas ; YANG, Fan ; DEMERS, Alan ; GEHRKE, Johannes ; RIEDEWALD, Mirek ; SHANMUGASUNDARAM, Jayavel: Automatic client-server partitioning of data-driven web applications. In: *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*. New York, NY, USA : ACM, 2006 (SIGMOD '06), S. 760–762. – URL http://doi.acm.org/10.1145/1142473.1142580. – ISBN 1-59593-434-0

[Giraldo u. a. 2007] GIRALDO, William J. ; ORTEGA, Manuel ; OLIVAS, José A.: A study about browsers in the Web and the Desktop. In: *Proceedings of the 2007 Euro American conference on Telematics and information systems*. New York, NY, USA : ACM, 2007 (EATIS '07), S. 23:1–23:8. – URL http://doi.acm.org/10.1145/1352694.1352718. – ISBN 978-1-59593-598-4

[Harjono u. a. 2010] HARJONO, Johan ; NG, Gloria ; KONG, Ding ; LO, Jimmy: Building smarter web applications with HTML5. In: *Proceedings of the 2010 Conference of the Center for Advanced Studies on Collaborative Research*. Riverton, NJ, USA : IBM Corp., 2010 (CASCON '10), S. 402–403. – URL http://dx.doi.org/10.1145/1923947.1924015

[Javadoc javafx.scene.Scene] *javadoc Class javafx.scene.Scene*. – URL http://docs.oracle.com/javafx/2/api/javafx/scene/Scene.html. – abgerufen am 11.5.2012

[Javadoc javafx.scene.web.WebEngine] *javadoc Class javafx.scene.web.WebEngine*. – URL http://docs.oracle.com/javafx/2/api/javafx/scene/web/WebEngine.html. – abgerufen am 11.5.2012

[Javadoc javafx.scene.web.WebView] *javadoc Class javafx.scene.web.WebView*. – URL http://docs.oracle.com/javafx/2/api/javafx/scene/web/WebView.html. – abgerufen am 11.5.2012

[junit] *JUnit - Resources for Test Driven Development*. – URL http://www.junit.org/. – abgerufen am 1.8.2012

[Kuuskeri und Mikkonen 2009] KUUSKERI, Janne ; MIKKONEN, Tommi: Partitioning web applications between the server and the client. In: *Proceedings of the 2009 ACM symposium on Applied Computing*. New York, NY, USA : ACM, 2009 (SAC '09), S. 647–652. – URL http://doi.acm.org/10.1145/1529282.1529416. – ISBN 978-1-60558-166-8

[Luxor] *Luxor XUL, History - What's New?*. – URL http://luxor-xul.sourceforge.net/history.html. – abgerufen am 11.5.2012

[McFall und Cusack 2009] MCFALL, Ryan ; CUSACK, Charles: Developing interactive web applications with the Google Web Toolkit. In: *J. Comput. Sci. Coll.* 25 (2009), Oktober, Nr. 1, S. 30–31. – URL http://dl.acm.org/citation.cfm?id=1619221.1619228. – ISSN 1937-4771

[Mickens und Dhawan 2011] MICKENS, James ; DHAWAN, Mohan: Atlantis: robust, extensible execution environments for web applications. In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. New York, NY, USA : ACM, 2011 (SOSP '11), S. 217–231. – URL http://doi.acm.org/10.1145/2043556.2043577. – ISBN 978-1-4503-0977-6

[Mikkonen und Taivalsaari 2008a] MIKKONEN, Tommi ; TAIVALSAARI, Antero: Towards a uniform web application platform for desktop computers and mobile devices. Mountain View, CA, USA : Sun Microsystems, Inc., 2008. – Forschungsbericht

[Mikkonen und Taivalsaari 2008b] MIKKONEN, Tommi ; TAIVALSAARI, Antero: Web Applications - Spaghetti Code for the 21st Century. In: *Proceedings of the 2008 Sixth International Conference on Software Engineering Research, Management and Applications*. Washington, DC, USA : IEEE Computer Society, 2008 (SERA '08), S. 319–328. – URL http://dx.doi.org/10.1109/SERA.2008.16. – ISBN 978-0-7695-3302-5

[Mikkonen und Taivalsaari 2011] MIKKONEN, Tommi ; TAIVALSAARI, Antero: Reports of the Web's Death Are Greatly Exaggerated. In: *Computer* 44 (2011), Mai, Nr. 5, S. 30–36. – URL http://dx.doi.org/10.1109/MC.2011.127. – ISSN 0018-9162

[mockito] *Mocktito - simpler and better mocking*. – URL http://code.google.com/p/mockito/. – abgerufen am 1.8.2012

[MSDN WebBrowser Class] *MSDN WebBrowser Class*. – URL http://msdn.microsoft.com/en-US/library/system.windows.controls.webbrowser.aspx. – abgerufen am 24.5.2012

[Nyrhinen und Mikkonen 2009] NYRHINEN, Feetu ; MIKKONEN, Tommi: Web Browser as a Uniform Application Platform: How Far Are We? In: *Proceedings of the 2009 35th Euromicro Conference on Software Engineering and Advanced Applications*. Washington, DC, USA : IEEE Computer Society, 2009 (SEAA '09), S. 578–584. – URL http://dx.doi.org/10.1109/SEAA.2009.37. – ISBN 978-0-7695-3784-9

[Obcena 2009] OBCENA, Mark: Rich cross-platform desktop applications using open-source Titanium. In: *Linux J.* 2009 (2009), September, Nr. 185. – URL http://dl.acm.org/citation.cfm?id=1610564.1610566. – ISSN 1075-3583

[Pemberton 2005] PEMBERTON, Steven: The future of web interfaces. In: *Proceedings of the 2005 IFIP TC13 international conference on Human-Computer Interaction*. Berlin, Heidelberg : Springer-Verlag, 2005 (INTERACT'05), S. 4–5. – URL http://dx.doi.org/10.1007/11555261_3. – ISBN 3-540-28943-7, 978-3-540-28943-2

[Pohja 2010] POHJA, Mikko: Comparison of common XML-based web user interface languages. In: *J. Web Eng.* 9 (2010), Juni, Nr. 2, S. 95–115. – URL http://dl.acm.org/citation.cfm?id=2011309.2011310. – ISSN 1540-9589

[Puder 2004] PUDER, Arno: Extending desktop applications to the web. In: *Proceedings of the 2004 international symposium on Information and communication technologies*, Trinity College Dublin, 2004 (ISICT '04), S. 8–13. – URL http://dl.acm.org/citation.cfm?id=1071509.1071512. – p8-puder.pdf. – ISBN 1-59593-170-8

[Shubin und Perkins 1998] SHUBIN, Hal ; PERKINS, Ron: Web navigation: resolving conflicts between the desktop and the Web. In: *CHI 98 conference summary on Human factors in computing systems*. New York, NY, USA : ACM, 1998 (CHI '98), S. 209–. – URL http://doi.acm.org/10.1145/286498.286694. – ISBN 1-58113-028-7

[swt] *Eclipse SWT Project Website*. – URL http://www.eclipse.org/swt/. – abgerufen am 24.5.2012

[Sykora 2004] SYKORA, Milan: *W3C-DOM Core Level 2*. 2004. – URL http://ics.upjs.sk/~jirasek/sps/dom/core%20level2.png. – abgerufen am 28.7.2012

[Taivalsaari und Mikkonen 2011] TAIVALSAARI, Antero ; MIKKONEN, Tommi: The Web as an Application Platform: The Saga Continues. In: *Proceedings of the 2011 37th EUROMICRO Conference on Software Engineering and Advanced Applications*. Washington, DC, USA : IEEE Computer Society, 2011 (SEAA '11), S. 170–174. – URL http://dx.doi.org/10.1109/SEAA.2011.35. – ISBN 978-0-7695-4488-5

[w3c-selectors-api2] HUNT, Lachlan: *Selector API Level 2*. – URL http://www.w3.org/TR/selectors-api2/. – abgerufen am 1.8.2012

[webkit] *The WebKit Open Source Project*. – URL http://www.webkit.org/. – abgerufen am 15.5.2012

[WebKit Source Code 2012] *WebKit Source Code, Header File webkitwebresource.h*. 2012. – URL http://trac.webkit.org/browser/trunk/Source/WebKit/gtk/webkit/webkitwebresource.h. – abgerufen am 15.5.2012

[Wikipedia AJAX 2012] *Ajax (programming)*. 2012. – URL http://en.wikipedia.org/wiki/Ajax_%28programming%29. – abgerufen am 1.8.2012

[xpcom] *Mozilla XPCOM*. – URL https://developer.mozilla.org/en/XPCOM. – abgerufen am 18.5.2012

[XUL] *XUL*. – URL https://developer.mozilla.org/En/XUL. – abgerufen am 24.5.2012

[XULWishlist] *XUL Wishlist*. – URL https://wiki.mozilla.org/XUL:Wishlist. – abgerufen am 24.5.2012

A. Anhang

A.1. Quelltext der Klasse ReceiptListViewImpl

```
1  public class ReceiptListViewImpl implements ReceiptListView {
2
3          private ReceiptListPresenter presenter;
4
5          private Group panel;
6
7          private  List  receiptListWidget;
8
9          public ReceiptListViewImpl(ReceiptListPresenter presenter) {
10                 this.presenter = presenter;
11         }
12
13         public Control  init (Composite parent, int style)  {
14
15                 this.panel = new Group(parent, SWT.NONE);
16                 GridLayout layout = new GridLayout();
17                 this.panel.setLayout(layout);
18
19                 Button button = new Button(panel, SWT.DEFAULT);
20                 button.setText("New");
21                 button.addSelectionListener(new SelectionAdapter() {
22                         @Override
23                         public void widgetSelected(SelectionEvent ev) {
24                                 onNewButtonClicked();
25                         }
26                 });
27
28                 receiptListWidget = new List(panel,  style );
29                 GridData layoutData = new GridData();
30                 layoutData.grabExcessHorizontalSpace = true;
31                 layoutData.grabExcessVerticalSpace = true;
32                 layoutData.horizontalAlignment = GridData.FILL;
33                 layoutData.verticalAlignment = GridData.FILL;
34                 receiptListWidget.setLayoutData(layoutData);
35
```

```
36                  receiptListWidget.addSelectionListener(new SelectionAdapter() {
37
38                      @Override
39                      public void widgetSelected(SelectionEvent ev) {
40                          int idx = receiptListWidget.getSelectionIndex();
41                          if (idx >= 0 && idx < receiptListWidget.getItemCount())
42                              presenter.receiptSelected(idx);
43                          ev.doit = false;
44                      }
45                  });
46
47                  return receiptListWidget;
48          }
49
50          private void onNewButtonClicked() {
51                  this.presenter.newReceiptButtonClicked();
52          }
53
54          @Override
55          public void updateReceiptList(java.util.List<Receipt> receipts,
56                  Receipt selectedReceipt) {
57
58                  for (int i = 0; i < receipts.size(); ++i) {
59                      Receipt receipt = receipts.get(i);
60                      if (i < receiptListWidget.getItemCount()) {
61                          receiptListWidget.setItem(i, receipt.getReceiptNumber());
62                      } else {
63                          receiptListWidget.add(receipt.getReceiptNumber());
64                      }
65
66                      if (receipt == selectedReceipt)
67                          receiptListWidget.select(i);
68                  }
69
70                  if (receipts.size() < receiptListWidget.getItemCount())
71                      receiptListWidget.remove(receipts.size(),
72                              receiptListWidget.getItemCount() - 1);
73          }
74  }
```

Listing A.1: ReceiptListViewImpl

Versicherung über Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 13. August 2012	
Ort, Datum	Unterschrift