



Abschlussprüfung Sommer 2018

Fachinformatiker für Anwendungsentwicklung  
Dokumentation zur betrieblichen Projektarbeit

## Shortcut-Editor

Implementierung eines Editors zur Bearbeitung von  
Tastaturkürzeln

Abgabetermin: 18.05.2018

Projektverantwortlicher: Robert Loipfinger

**Prüfungsbewerber:**

Korbinian Mifka  
Pelzgartenstraße 12  
84175 Johannesbrunn



**Ausbildungsbetrieb:**

ADITO Software GmbH  
Konrad Zuse Str. 4  
84144 Geisenhausen

Dieses Werk einschließlich seiner Teile ist **urheberrechtlich geschützt**. Jede Verwertung außerhalb der engen Grenzen des Urheberrechtsgesetzes ist ohne Zustimmung des Autors unzulässig und strafbar.



## Inhaltsverzeichnis

<b>1</b>	<b>Einleitung</b>	<b>1</b>
1.1	Beschreibung . . . . .	1
1.2	Ziel . . . . .	1
1.3	Umfeld . . . . .	1
1.4	Begründung . . . . .	2
1.5	Schnittstellen . . . . .	2
1.6	Abgrenzung . . . . .	2
<b>2</b>	<b>Projektplanung</b>	<b>3</b>
2.1	Entwicklungsprozess . . . . .	3
2.2	Projektphasen . . . . .	3
2.3	Ressourcenplanung . . . . .	3
<b>3</b>	<b>Analysephase</b>	<b>4</b>
3.1	Ist-Analyse . . . . .	4
3.2	Sollkonzept . . . . .	4
3.3	Anwendungsfälle . . . . .	4
3.4	„Make or Buy“-Entscheidung . . . . .	4
3.5	Lastenheft . . . . .	4
<b>4</b>	<b>Entwurfsphase</b>	<b>5</b>
4.1	Architekturdesign . . . . .	5
4.2	Benutzeroberfläche . . . . .	6
4.3	Datenmodell für Entitäten und Funktionen . . . . .	7
4.4	Datenmodell für Browsertestergebnisse . . . . .	8
4.5	Datenstore . . . . .	8
4.6	Pflichtenheft . . . . .	8
<b>5</b>	<b>Implementierung</b>	<b>9</b>
5.1	Back-End . . . . .	9
5.1.1	Datenmodelle . . . . .	9
5.1.2	Datenstore . . . . .	9
5.2	Front-End . . . . .	9
5.2.1	ShortcutField . . . . .	9
5.2.2	Accordion . . . . .	9
5.2.3	Check-Button . . . . .	10
5.2.4	CheckItemContainer . . . . .	10
<b>6</b>	<b>Anwendungstests</b>	<b>11</b>
<b>7</b>	<b>Fazit</b>	<b>13</b>
<b>8</b>	<b>Glossar</b>	<b>14</b>
<b>9</b>	<b>Abbildungsverzeichnis</b>	<b>15</b>
<b>10</b>	<b>Literaturverzeichnis</b>	<b>15</b>
<b>11</b>	<b>Anhang</b>	<b>16</b>



# 1 Einleitung

Die folgende Projektdokumentation beschreibt den Ablauf des IHK-Abschlussprojektes, welche der Autor im Rahmen der Ausbildung zum Fachinformatiker Fachrichtung Anwendungsentwicklung durchgeführt hat. Ausbildungsbetrieb ist die ADITO Software GmbH, ein Hersteller für hochflexible Business-, CRM- und xRM-Software mit Sitz in Geisenhausen. Das inhabergeführte Unternehmen bietet Entwicklung, Vertrieb, Projektierung und Service aus einer Hand. Kunden von ADITO kommen aus den unterschiedlichsten Branchen. Neben namhaften mittelständischen Unternehmen, zum Beispiel Ravensburger, Erlus oder Birco, gehören auch große Organisationen wie die WWK Versicherungsgruppe oder die Bundesagentur für Arbeit zu ihren Referenzen.

## 1.1 Beschreibung

In der neuesten Version des xRM-Systems ADITO5 wurde eine neue Clientvariante entwickelt. Nun ist es möglich neben dem konventionellen Java Swing Client, einen Webclient bzw. Browserclient einzusetzen. Dieser bietet einige Vorteile. Beispielsweise ist keine Installation notwendig und die Nutzung ist auf allen Geräten mit Webbrowsern (PC, Tablet und Smartphone) möglich.

Mit einem Webclient gehen allerdings auch einige Herausforderungen einher. So auch bei der Vergabe von Tastaturkürzeln. Browser behalten es sich vor, einige Shortcuts für eigene Aktionen zu reservieren und so nicht für die eigentliche Webanwendung zur Verfügung zu stellen. Beispiele für solche Shortcuts sind Strg + P für Drucken oder Strg + F für Suchen. Der Überblick über die Verwendbarkeit von Tastaturkürzeln geht schnell verloren, da diese in jeder Browser-Betriebssystem permutation variieren kann.

In diesem Projekt soll eine Möglichkeit geschaffen werden, bei der Vergabe von Shortcuts innerhalb der hauseigenen xRM-Entwicklungsumgebung (ADITO-Designer) Unterstützung zu bieten.

## 1.2 Ziel

Um den Projektierern unserer xRM-Software die Vergabe von Shortcuts zu erleichtern soll ein spezieller Shortcut-Editor entwickelt werden, der die Eingabe und Bearbeitung von Shortcuts ermöglicht und bei der Wahl des passenden Tastenkürzels zuarbeitet.

Mittels Warnungen im Editor soll verdeutlicht werden, dass der eingegebene Shortcut zu Problemen auf einem bestimmten Browser führen kann. Damit der Benutzer feststellen kann, warum ein Shortcut problematisch ist, sollen weitere Informationen angezeigt werden. Diese können beispielsweise angeben, in welchem Browser bzw. welcher Version das Tastaturkürzel bereits verwendet wird.

## 1.3 Umfeld

Durchgeführt wird das Projekts in der Abteilung Entwicklung, welche auch für die Umsetzung von ADITO5 zuständig war. Im Zuge der Weiterentwicklung wurde innerhalb der Entwicklungsabteilung die Notwendigkeit des Editors festgestellt. Dadurch kann man die Abteilung Entwicklung selbst als Auftraggeber ihres eigenen Projekts ansehen.

Die Implementierung des Editors wird in der objektorientierten Programmiersprache Java und mithilfe der Entwicklungsumgebung IntelliJ IDEA durchgeführt. Als Framework für die GUI dient das bekannte Java-Swing-Framework.



## 1.4 Begründung

Da gewährleistet werden soll, dass vergebene Tastenkürzel auf allen relevanten Browsern funktionieren, muss dem Projektierer bei der Wahl eines passenden Shortcuts immer klar sein, ob dieser von den entsprechenden Browsern unterstützt wird. Da jeder Browser andere Shortcuts vorbelegt und diese sich je nach Betriebssystem wieder unterscheiden können, ist eine manuelle Überprüfung durch den Projektierer so gut wie unmöglich. So müsste dieser auf jedem Betriebssystem alle relevanten Browser testen. Ein solch enormer Aufwand und die mögliche Fehlvergabe von Tastenkombinationen kann durch die technische Assistenz mittels des genannten Editors vermieden werden.

## 1.5 Schnittstellen

Um herauszufinden, welche Shortcuts die verschiedenen Browser auf unterschiedlichen Betriebssystemen selber verwenden, wurde außerhalb dieser Abschlussarbeit eine Testanwendung implementiert. Diese läuft auf jeder Plattform und testet alle möglichen Shortcut-Kombinationen für die verbreitetsten Browser. Das Ergebnis dieser Tests wird in Form von XML-Dateien gespeichert (Beispiel siehe Anlage (XXX)). Für jede Browser-Betriebssystem Kombination existiert eine eigene Datei, in welcher alle problembehafteten Shortcuts verzeichnet sind.

Damit die in den Dateien enthaltenen Informationen dem Benutzer dargestellt werden können, muss der Editor das Einlesen und Verarbeiten von XML-Strukturen beherrschen. Hierfür kommt das hauseigene Property Framework zum Einsatz. Dieses kümmert sich um sämtliche XML-spezifische Arbeiten und ermöglicht so eine komfortable Nutzung.

Um das Ergebnis des Editors im bestehenden ADITO Designer einfach verwenden zu können, muss dieses als IShortcut-Typ zurückgegeben werden. Dieser ADITO eigene Datentyp wird im restlichen System bereits für Shortcuts verwendet und bietet sich somit an.

## 1.6 Abgrenzung

Aufgrund des beschränkten Projektumfangs ist das Einbinden des Editors in den bestehenden ADITO Designer nicht Bestandteil der Projektarbeit.

## 2 Projektplanung

### 2.1 Entwicklungsprozess

Um das Projekt realisieren zu können, musste sich für einen geeigneten Entwicklungsprozess entschieden werden. Dieser gibt die Vorgehensweise vor, welche der Umsetzung zu Grunde liegt. Für dieses Projekt wurde vom Autor das Wasserfallmodell gewählt. Dabei wird die Umsetzung auf 5 Phasen aufgeteilt (siehe Abbildung 1): Die Ermittlung der Anforderungen, die Erstellung eines Entwurfs, die Implementierung und am Ende die Überprüfung und Wartung der erstellten Software. Dieses Modell bietet sich für diese Arbeit an, da die Anforderungen an den Editor klar definiert sind und sich während der Umsetzungsphase nicht ändern.

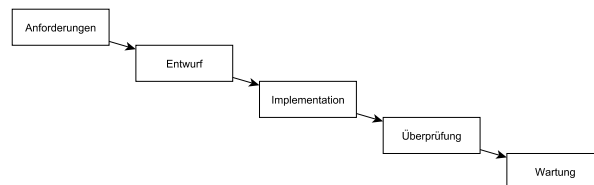


Abbildung 1: Wasserfallmodell

### 2.2 Projektphasen

Zur Realisierung des Abschlussprojekts standen insgesamt 70 Stunden zur Verfügung. Diese Zeit wurde vor Projektbeginn auf verschiedene Phasen verteilt, die während der Durchführung durchlaufen werden. Diese Zeitplanung der Hauptphasen lässt sich aus Abbildung 2 entnehmen.

Vorgang	Geplante Zeit in h
1. Analysephase	3
2. Entwurfsphase	15
3. Implementierungsphase	40
4. Testphase	2
5. Dokumentationserstellung	10
	70

Abbildung 2: Grobe Zeitplanung

### 2.3 Ressourcenplanung

Im Zuge der Ressourcenplanung wurde eine Übersicht (siehe Anhang Unterabschnitt 11.2) erstellt. Diese enthält sämtliche Ressourcen, welche innerhalb der Durchführung des Projekts eingesetzt wurden. Dabei handelt es sich sowohl um Hard- und Softwareressourcen als auch um Personal. Zur Minimierung der Projektkosten wurde bevorzugt kostenfreie Software verwendet. War dies nicht möglich, so wurde Software eingesetzt, für welche die ADITO Software GmbH bereits Lizenzen besaß.

## 3 Analysephase

### 3.1 Ist-Analyse

Seit früheren Versionen existierte bereits ein Editor zur Eingabe von Shortcuts (siehe Abbildung 3). Dieser ist allerdings sehr einfach aufgebaut und beschränkt sich auf die Eingabe eines Shortcuts per Tastatur. Außerdem ist es nicht möglich Warnungen anzuzeigen oder zwischen bestehenden Shortcuts zu navigieren. Da dieser Editor in keinerlei Hinsicht den gegebenen Anforderungen dieses Projekts entspricht, wurde eine Weiterentwicklung dessen vom Autor als nicht sinnvoll erachtet.

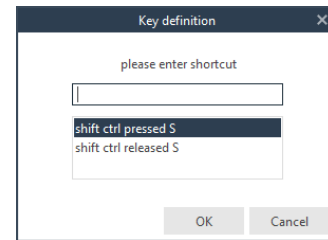


Abbildung 3: Bestehender Editor

Die Funktionen, welchen Shortcuts zugeordnet werden sollen sind im ADITO Designer schon vorhanden. Zudem existieren sogenannte Entitys, welche Dateneinheiten darstellen und eben genannte Funktionen besitzen können. Beispielsweise könnte ein Entity „Person“ existieren, welche wiederum die Funktion „Person hinzufügen“ besitzt. Dieser Funktion könnte nun ein Shortcut zugewiesen werden z.B. Strg + Einfg.

### 3.2 Sollkonzept

Der neue Editor muss ebenfalls die Eingabe aber auch die Bearbeitung (z.B. Entfernen einer einzelnen Taste) eines Shortcuts per Tastatur und Maus unterstützen. Für die Navigation und für einen besseren Überblick, werden alle bestehenden Tastenkombinationen in tabellarischen Form präsentiert. Es soll zu jedem Zeitpunkt ersichtlich sein, für welche Funktion der Shortcut definiert wird. Eine weitere Anforderung besteht darin, alle Warnungen für die entsprechenden Browser und deren Betriebssysteme anzuzeigen.

### 3.3 Anwendungsfälle

Um eine grobe Übersicht über alle Anwendungsfälle zu erhalten, die von dem umzusetzenden Editor abgedeckt werden sollen, wurde im Laufe der Analysephase ein Use-Case-Diagramm erstellt. Dieses Diagramm befindet sich im Anhang Abbildung 20 auf Seite 16.

### 3.4 „Make or Buy“-Entscheidung

Die Entscheidung, ob der Editor selber erstellt oder gekauft werden soll, lässt sich einfach treffen. Sucht man auf dem Markt nach Softwareteilen, welche den Anforderungen dieses Projekts genüge tun, so findet man nichts entsprechendes. Darum ist man gewissermaßen gezwungen den Editor selbst zu erstellen.

### 3.5 Lastenheft

Basierend auf dem Sollkonzept und den Anwendungsfällen wurde am Ende der Analysephase ein Lastenheft erarbeitet. Dieses umfasst alle Anforderungen, welche an den Editor gerichtet werden. Ein Auszug aus dem Lastenheft befindet sich im Anhang (XXX) auf S. (X)

## 4 Entwurfsphase

### 4.1 Architekturdesign

Als passendes Entwurfsmuster für den Shortcut Editor hat sich das Model View Presenter (MVP)-Architekturmuster herausgestellt. Dieses ist nicht ganz so verbreitet wie das bekanntere Model View Controller (MVC)-Muster, ist diesem aber sehr ähnlich. Der eigentliche Unterschied zwischen MVP und MVC liegt darin, dass bei MVC die View neben dem Controller auch mit dem Model kommuniziert und dieses somit kennen muss. Bei MVP ist die View völlig unabhängig vom Model und nur der Presenter kommuniziert mit ihr (siehe Abbildung 4). Der Autor hat sich für das MVP-Entwurfsmuster entschieden, da damit die View auch ohne Model verwendet werden kann und es denkbar ist, dass diese Möglichkeit in Zukunft benötigt wird.

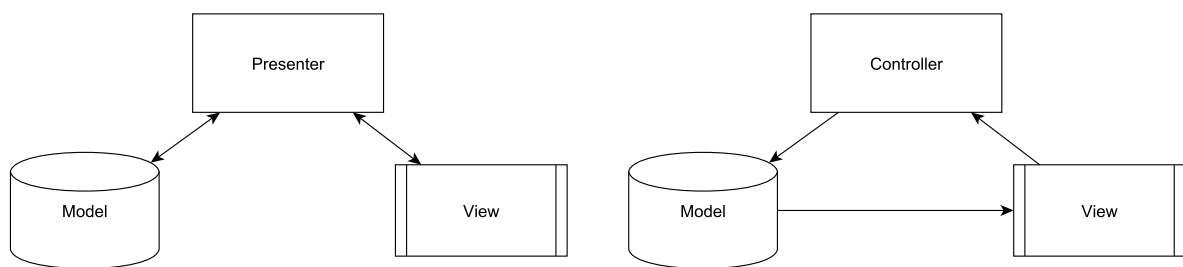


Abbildung 4: Model View Presenter vs. Model View Controller

Bei MVP lässt sich jede Komponente der Software einem der drei Bestandteile - Model, View oder Presenter - zuordnen. Jeder dieser Teile hat einen eigenen Aufgabenbereich, der von denen der anderen weitestgehend unabhängig ist. Im Model werden alle Daten gehalten und zur Verfügung gestellt. Die View kümmert sich um die grafische Darstellung und der Presenter stellt das Bindeglied zwischen der View und dem Model dar. Verändern sich beispielsweise die Werte in der View, so kümmert sich der Presenter darum, dass diese Wertänderung im Model ebenso stattfindet und andersherum. Der Presenter hält somit die View – oder auch mehrere Views untereinander – und das Model synchron zueinander. Die lose Kopplung der einzelnen Komponenten erhöht die Wiederverwendbarkeit und Austauschbarkeit. Man könnte beispielsweise die Benutzeroberfläche austauschen, ohne das Model anpassen zu müssen. Außerdem können die einzelnen Komponenten durch die strikte Trennung einfacher getestet, gewartet und flexibel erweitert werden. Diese Vorteile sprechen ebenso für eine Verwendung von MVP.

Im Sinne der Wiederverwendbarkeit, werden auch alle GUI-Komponenten des Shortcut Editors völlig separat voneinander und unabhängig vom Editor implementiert. Dadurch kann gewährleistet werden, keine unnötigen Abhängigkeiten zu editorspezifischen Teilen aufzubauen. So ist die Benutzung von Komponenten auch an anderen Stellen in der Software ohne weiteren Aufwand möglich.

Wie im Abschnitt 1.5 bereits erwähnt wird für das Lesen der Testergebnis XML-Dateien das haus eigene Property Framework verwendet. Dieses Tool stellt Funktionalitäten zum Lesen und Schreiben von XML zur Verfügung. Außerdem kümmert es sich eigendständig um die Konvertierung von Datentypen. Dadurch kommt der Autor bei der Implementierung nicht mit XML spezifischen Arbeiten in Berührung und kann sich auf den eigentlichen Editor konzentrieren.

## 4.2 Benutzeroberfläche

Für das Design des User Interfaces (UI) wurden von einem UX-Designer der ADITO Software GmbH Entwürfe angefertigt (siehe Abbildung 5). Im Designentwurf wird ersichtlich, welche Komponenten verwendet werden müssen, um alle angeforderten Informationen darzustellen und wie diese aussehen und angeordnet zu sein haben, um die bedarfsgerechte Bedienung zu ermöglichen. Nachfolgend werden die Bestandteile des Editors näher erläutert:

- ① **Breadcrumb:** Diese Komponente ist in der Lage einen Pfad darzustellen und diesen zu bearbeiten. In diesem Fall besteht sie aus beliebig vielen ComboBoxen, um an jeder Stelle des Pfads einen anderen Knoten auswählen zu können. Diese Komponente dient zum einen als Orientierungshilfe, um jederzeit feststellen zu können, für welche Funktion der Shortcut gesetzt wird. Zum Anderen ist damit eine intuitive Navigation durch alle Funktionen möglich.
- ② **Shortcut-Field:** Hierbei handelt es sich um eine Komponente, welche die Darstellung und Bearbeitung von Shortcuts ermöglicht. Die Eingabe und Editierung des Tastaturkürzels kann nur bei fokussiertem Zustand erfolgen. Demnach wechselt die Komponente bei Fokussierung in den Bearbeitungsmodus und verlässt diesen, sobald eine andere Komponente den Fokus erlangt.
- ③ **Check-Button:** Dieses GUI-Element kann selektiert werden und stellt neben einem Icon ein Häkchen- oder X-Symbol dar. Somit kann die Komponente visualisieren, bei welchem Browser bzw. Betriebssystem der Shortcut Probleme bereiten kann (X) und wo dieser unbedenklich ist (Häkchen). Außerdem dient sie dem Benutzer zur Auswahl eines Elements, um davon mehr Informationen zu erhalten (Im Entwurf werden beispielhaft für Google Chrome und macOS detaillierte Informationen angezeigt).
- ④ **Shortcut-Tag:** Ein Shortcut-Tag dient zur Darstellung einer Tastenkombination und bietet die Möglichkeit sich mittels eines X-Buttons selber zu entfernen.
- ⑤ **TreeTable:** Zur Darstellung der zugrundeliegenden Baumstruktur wird zur Visualisierung aller Funktionen eine Kombination aus Tree und Tabelle verwendet. Sie dient – wie die **Breadcrumb** – der Navigation und bietet zudem einen Überblick über alle vorhandenen Shortcuts.
- ⑥ **Accordion:** Ist ein **Check-Button** selektiert, so wird eine Accordion-Komponente angezeigt, welche detaillierte Informationen zu den Testergebnissen bietet. Um nur relevante Daten anzuzeigen, besteht die Möglichkeit einige Sektionen durch Klicken auf den Header einzuklappen.

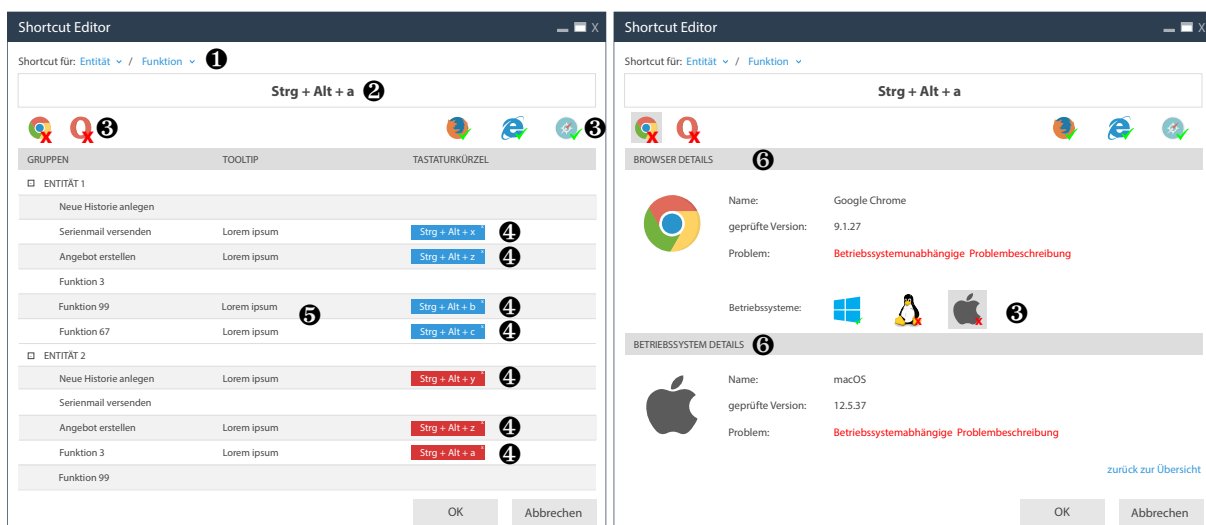


Abbildung 5: Designentwurf des UX-Designers



### 4.3 Datenmodell für Entitäten und Funktionen

Wie sich im Designentwurf (Abbildung 5) – aufgrund der TreeTable – schon erahnen lässt, sollen die Entitäten und deren Funktionen als Baumstruktur aufgebaut werden. Jede Funktion stellt einen Endknoten (Blatt) dar und soll genau einen Shortcut besitzen können. Entitäten hingegen ist es nur erlaubt, Funktionen und weitere Entitäten aufzunehmen (siehe Abbildung 6). Da es sich um eine Baumstruktur handelt, kann man ausschließen, dass sich eine Entität selbst als Kind hält.

Um diese Struktur im Editor abbilden zu können, wurde ein Datenmodell entworfen, welches den Anforderungen entspricht. Zur Erläuterung des Modells ist im Folgenden ein schematisches UML Klassendiagramm abgebildet, welches den Grundaufbau und die Beziehungen zwischen den Elementen verdeutlicht (Abbildung 7).

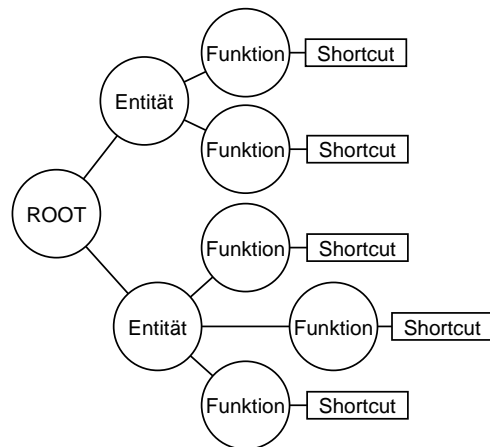


Abbildung 6: Baumstruktur für Entitäten und Funktionen

Zentraler Bestandteil des Datenmodells ist das Interface **INode**. Dieses kann neben einem eigenen Namen auch seine Kinder zurückgeben. Diese sind ebenfalls vom Typ **INode**. Damit kann grundsätzlich schon eine Baumstrukturen aufgebaut werden. Allerdings ist durch **INode** nur die Abbildung der Bestandteile **ROOT** und **Entity** aus Abbildung 6 möglich, da noch kein Tastenkürzel gehalten werden kann.

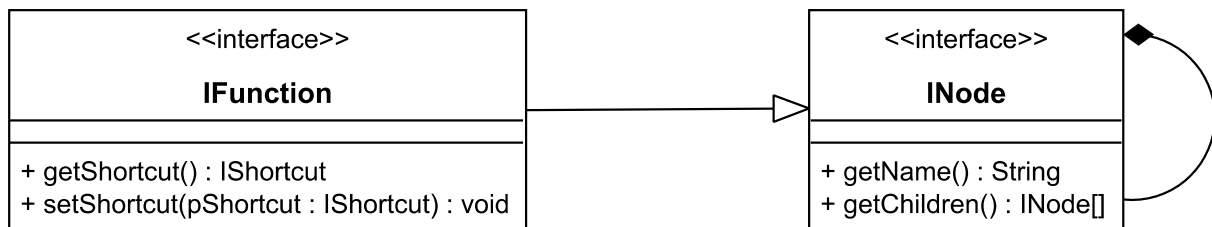


Abbildung 7: Datenmodell für Entitäten und Funktionen

Um auch eine Funktion im Modell abbilden zu können, existiert das Interface **IFunction**. Dieses erbt von **INode** und stellt somit einen vollwertigen Knoten dar, welcher bei der Methode **getChildren()** zurückgegeben werden kann. Über die Methoden **setShortcut(...)** kann eine Tastenkombination gesetzt werden und **getShortcut()** ermöglicht das Auslesen. Da diese Methoden nur die Zuweisung von einem einzigen Shortcut zulassen, ist sichergestellt, dass eine Funktion nur ein Tastenkürzel besitzen kann. Aufgrund der Tatsache, dass **IFunction** einen Endknoten (Blatt) darstellt und somit keine Kinder hat, liefert die geerbte Methode **getChildren()** ein leeres Array. Für die Iteration durch den Baum ist es programmatisch komfortabler und effizienter, wenn jeder Knoten die Methode **getChildren()** besitzt, da andernfalls unnötige Abfragen stattfinden müssen.

Über diese Konstellation lässt sich die Baumstruktur der Funktionen und deren Shortcuts den Anforderungen entsprechend abbilden.

#### 4.4 Datenmodell für Browsertestergebnisse

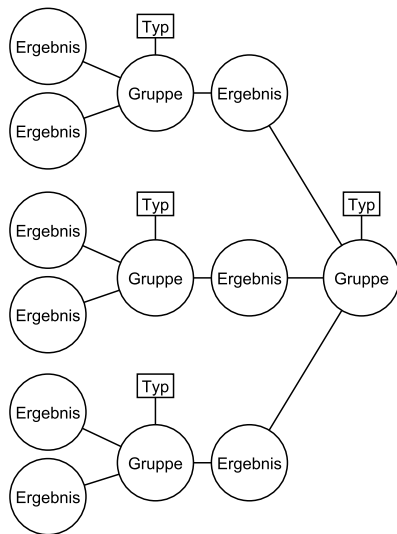


Abbildung 8: Baumstruktur für  
Browsertestergebnisse

Auch die Browserergebnisse werden als Baumstruktur gehalten. Allerdings ergibt sich hierbei eine neue Anforderung. Betrachtet man den Designentwurf, so stellt man fest, dass eine Gruppe von Testergebnissen immer eine Typenbezeichnung hat (Im Entwurf haben die Gruppen den Typ „Browser“ oder „Betriebssystem“). In Abbildung 9 ist ein UML-Klassendiagramm eines Datenmodells abgebildet, welches den Anforderungen zum Halten von Browsertestergebnissen gerecht wird.

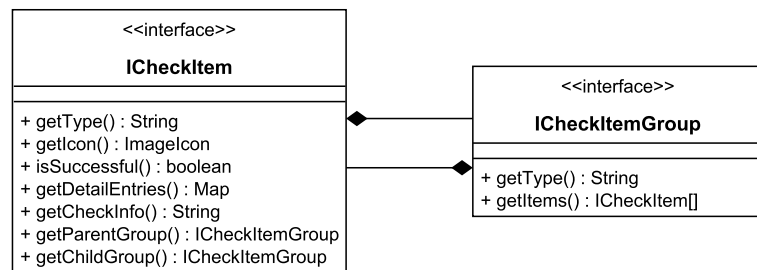


Abbildung 9: Datenmodell für  
Browsertestergebnisse

In diesem Modell stellt ein **ICheckItem** ein einzelnes Testergebnis dar. Eine **ICheckItemGroup** hält neben einer beliebigen Anzahl von **ICheckItems** einen Typ in Form eines **Strings**. Somit kann zu jeder Gruppe die gewünschte Typenbezeichnung hinzugefügt werden. Ein **ICheckItem** besitzt eine Parent- und ein Childgruppe. Diese können mittels der Methoden **getParentGroup** und **getChildGroup** erlangt werden. Über diese Methoden lässt sich die Baumstruktur aufbauen.

#### 4.5 Datenstore

Damit sowohl die Daten der Entitäten und Funktionen als auch die der Browsertestergebnissen in zuvor geschilderter Form erhalten werden können, soll ein Datenstore zum Einsatz kommen. Dieser liefert die beiden oben beschriebenen Datenmodelle. Dabei kümmert er sich die Daten von anderen Quellen (z.B. XML-Dateien) in die gewünschte Datenmodell-Form zu bringen. Zudem ist er dafür zuständig, die geänderten Daten wieder zurück zu speichern.



Abbildung 10: Datenstore

#### 4.6 Pflichtenheft

Um festzusetzen, wie die Anforderungen an den Editor vom Autor umgesetzt werden sollen, wurde am Ende der Entwurfsphase ein Pflichtenheft erstellt, welches auf dem Lastenheft basiert. Dieses dient somit als Vorgabe für die Realisierung des Projekts. Ein Auszug aus dem Pflichtenheft befindet sich im Anhang (XXX) auf S.(X)



## **5 Implementierung**

### **5.1 Back-End**

#### **5.1.1 Datenmodelle**

#### **5.1.2 Datenstore**

### **5.2 Front-End**

Nachfolgend wird die Implementierung der GUI beschrieben. Da einige Komponenten schon existiert haben und für den Editor nicht extra

#### **5.2.1 ShortcutField**

#### **5.2.2 Accordion**

### 5.2.3 Check-Button

Zur Implementierung des im Abschnitt (XXX) erwähnten Check-Buttons wurde zunächst das Interface `ICheckComponent` erstellt. Indessen wird die Grundfunktionalität einer `CheckComponent` definiert. Diese besteht darin, dass der Checked-Zustand gesetzt (`setChecked(...)`) und gelesen (`isChecked()`) werden kann. Außerdem informiert ein Listener über Änderungen des Zustands. Wird Checked auf true gesetzt, so soll innerhalb der Komponente ein grüner Hacken andernfalls ein rotes Kreuz angezeigt werden.

Die Klasse `CheckToggleButton` implementiert das beschriebene Interface und erbt von der Swing Klasse `JToggleButton`. Sie kümmert sich um das Einfügen und Aktualisieren des richtigen Symbols. Innerhalb von `setChecked(...)` wird die private Methode `_updateIcon()` aufgerufen, welche ihrerseits das jeweilige Symbol über das gesetzte Icon zeichnet (Siehe Anhang (XXX)). Insgesamt stellt ein `CheckToggleButton` eine vollwertige `CheckComponent` dar, welche direkt verwendet werden kann.

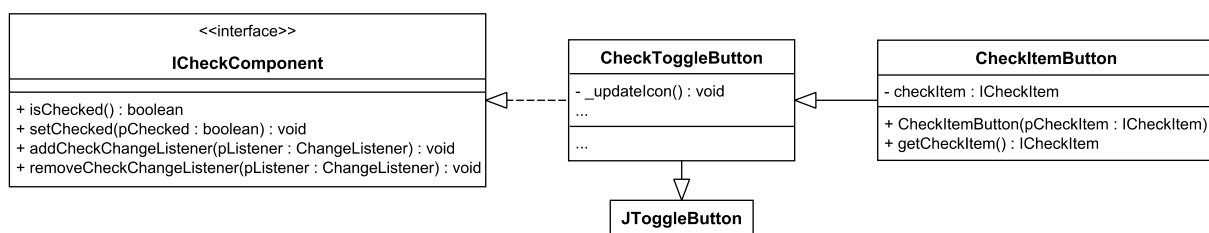


Abbildung 11: CheckItemButton

Um die Benutzung von `CheckToggleButtons` innerhalb des Shortcut Editors einfacher zu gestalten, wurde die Klasse `CheckItemButton` eingeführt, welche `CheckToggleButton` erweitert und über den Konstruktor ein `ICheckItem` (siehe (XXX)) aufnehmen kann. Innerhalb dieses Konstruktors werden dem Button alle Eigenschaften entsprechend dem `ICheckItem` gesetzt (z.B. Checked-Zustand oder Icon).

### 5.2.4 CheckItemContainer

Der Designentwurf lässt erkennen, dass die Check-Buttons einer bestimmten Sortierung unterliegen: auf der linken Seite befinden sich alle unchecked und auf der rechten Seite alle checked Buttons. Zudem wird bei den Browser Check-Buttons ein Abstand zwischen den beiden Buttongruppen eingefügt.

Um diese Positionierung zu realisieren, wurde der `CheckButtonContainer` eingeführt. Ihm lassen sich `CheckButtons` hinzufügen. Außerdem kann der Abstand über `setSplitting(...)` (de)aktiviert werden. Ändert sich der Zustand eines Check-Buttons, so kümmert er sich um die richtige Umsortierung.

Um wiederum die Nutzung innerhalb des Shortcut Editors zu erleichtern, existiert der `CheckItemContainer`. Diesem kann man einfache `CheckItems` übergeben (`setCheckItems(...)`), wonach dieser intern die benötigten `CheckButtons` erzeugt und sich selbst hinzufügt oder unnötige wieder entfernt. Zudem kann über die Methode `getSelection()` der aktuell selektierte `CheckButton` ausgelesen werden.

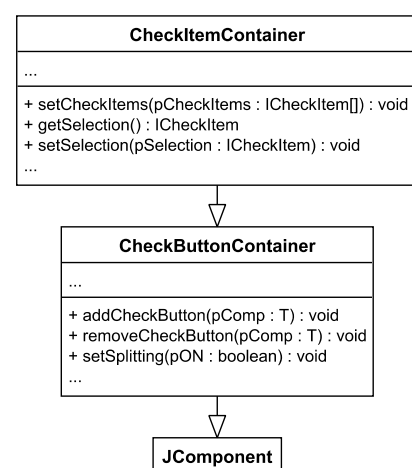


Abbildung 12:  
CheckItemContainer

## 6 Anwendungstests

Da das Projekt im Wasserfallmodell entwickelt wurde, fand gegen Ende ein umfangreiches Testen aller Teilmodule des Loggers statt. Getestet wurde mit dem Java-Framework „JUnit“.

JUnit ist ein einfaches, quelloffenes Framework zum Testen von Java-Programmen, das besonders für das automatisierte Testen einzelner Module geeignet ist. Es bietet ebenso ein leicht integrierbares Maven-Plugin, wodurch es sich perfekt in den Build-Vorgang von ADITO einfügt. Dieses Plugin führt alle vorhandenen Tests automatisch bei einem Kompilervorgang mit Maven aus. Dadurch fallen etwaige Programmierfehler sehr früh auf und können schon bei der Implementierung behoben werden.

Der JUnit-Test des Remote-Loggers ist in drei Teile unterteilt:

### *Initialisierung der Testkomponenten, (@Before)*

```

19 @Before
20 public void init() throws Exception
21 {
22     logger = new RemoteLogger('Z', "localhost",
23                               7733, _DummyFacade::new);
24     ...
25 }

```

Abbildung 13: Initialisierung des Remote-Logger-Servers

In der „init()“-Methode wird anfangs der Remote-Logger-Server mit der niedrigsten Priorität (Z) initialisiert. Ebenso wird die Adresse (localhost, Port 7733) bestimmt, auf die der Remote-Logger-Server hören soll.

Um den korrekten Login eines Remote-Logger-Clients am Remote-Logger-Server zu testen wird eine spezielle Implementierung des Interfaces „IRemoteLoggerLoginFacade“ (siehe ??) benötigt. Diese soll einen Verbindungsaufbau des Clients nur zulassen, wenn die empfangenen Login-Informationen „USER“ und „PASS“ enthalten. Das stellt eine Anmeldung mit Benutzername und Passwort dar.

```

133 @Override
134 public boolean checkLogin(String[] pLoginInformation)
135 {
136     return pLoginInformation.length == 2 &&
137           pLoginInformation[0].equals("USER") &&
138           pLoginInformation[1].equals("PASS");
139 }

```

Abbildung 14: Validieren des Logins

### *Hauptteil, (@Test)*

Im Hauptteil des JUnit-Tests, repräsentiert durch die „test-communication()“-Methode, wird die Funktionalität des Remote-Loggers auf die Probe gestellt.

Kurz zusammengefasst: Es werden zwei Remote-Logger-Clients erzeugt. Diese verbinden sich mit dem vorher initialisierten Remote-Logger-Server und erhalten CheckPoints. Das Auswerten empfangener Meldungen übernimmt die JUnit-Klasse „Assert“.

Die genaue Funktionsweise wird im Nachfolgenden erklärt:

Zu Anfang werden zwei Remote-Logger-Clients mit unterschiedlicher Sprache gestartet. Dafür wird eine Implementierung des Interfaces „IRemoteLoggerClientConnectionManager“ benötigt, welche durch die

```

146 private static class _ConnectionManager
147     extends AbstractRemoteLoggerClientConnectionManager
148 {
149     @Override
150     protected IRemoteLoggerClientConnection createConnection()
151         throws AditoException
152     {
153         return new RemoteLoggerClientConnection("localhost",
154                                                  7733, new String[]{"USER", "PASS"});
155     }
156 }

```

Abbildung 15: Verbindungsaufbau des Remote-Logger-Clients zum Remote-Logger-Server (Abbildung 15) abgebildet ist. Diese erhält passenden Login-Informationen und die Verbindungsparameter, um sich erfolgreich mit dem vorher gestarteten Remote-Logger-Server

Remote-Logger-Server zu verbinden.

Um auf Ereignisse des Remote-Logger-Servers zu reagieren, wird pro Remote-Logger-Client eine neue Instanz des Listeners „RemoteListener“ erzeugt und registriert. Dieser speichert empfangene CheckPoints in einer im Konstruktor übergebenen AtomicReference („refToSet“) und benachrichtigt anschließend alle Threads, die auf das Setzen dieser Referenz warten.

```

172 @Override
173 public void checkPointReceived(
174     @NotNull IRemoteLoggerCheckPoint pCheckPoint)
175 {
176     synchronized (refToSet)
177     {
178         refToSet.set(pCheckPoint);
179         refToSet.notifyAll();
180     }
181 }

```

Abbildung 16: Setzen eines empfangenen CheckPoints; Benachrichtigung der wartenden Threads

```
48 CPH.checkPoint(0, 1);
```

Abbildung 17: CheckPoint senden

Nachdem die Remote-Logger-Clients mit dem Remote-Logger-Server erfolgreich verbunden sind kann nun begonnen werden, CheckPoints zu senden. Hierzu wird der bereits vorhandene, ADITO4-eigene, „CheckPointHandler“ verwendet, der systemweit alle aufgetretenen CheckPoints an vorhandene Loggerimplementierungen übergibt.

Anschließend wird mit Hilfe der Methode „\_getNextCheckPoint(...)“ der zuletzt empfangene CheckPoint ausgelesen. Die übergebene AtomicReference dient hierfür als Container. Ist bereits ein Wert innerhalb dieses Containers gespeichert, dann wird dieser ausgelesen und zurückgegeben. Falls nicht wird so lange gewartet, bis der aktuelle Thread über das AtomicReference-Objekt benachrichtigt wird. Das Benachrichtigen erfolgt durch die Methode „notifyAll()“, die in der „checkPointReceived(...)“-Methode des registrierten Remote-Logger-Listeners aufgerufen wird (siehe Abbildung 16). Anschließend werden die Werte des empfangenen, deutschen CheckPoints auf Richtigkeit geprüft. Falls hierbei kein Fehler aufgetreten ist, wird der englische CheckPoint mit dem gleichen Verfahren überprüft. Hierfür muss nur die dahinterliegende Nachricht angepasst werden, denn ID, Modulnr., Programmnr., Typ und Priorität bleiben selbstverständlich gleich.

```

89 private IRemoteLoggerCheckPoint _getNextCheckPoint(
90     final AtomicReference<IRemoteLoggerCheckPoint>
91     pRefToWaitOn)
92 {
93     if(pRefToWaitOn.get() == null)
94     {
95         synchronized (pRefToWaitOn)
96         {
97             try
98             {
99                 pRefToWaitOn.wait();
100             }
101             catch (InterruptedException ignored)
102             {
103             }
104         }
105     }
106
107     IRemoteLoggerCheckPoint cp =
108         pRefToWaitOn.getAndSet(null);
109     Assert.assertNotNull(cp);
110     return cp;
111 }

```

Abbildung 18: Warten auf neue CheckPoints; Auslesen von CheckPoints

```

48 CPH.checkPoint(0, 1);
49 IRemoteLoggerCheckPoint cp = _getNextCheckPoint(lastCheckPointGER);
50 Assert.assertEquals(cp.getModule(), 0);
51 Assert.assertEquals(cp.getID(), 1);
52 Assert.assertEquals(cp.getMessage(),
53     "Interner Fehler. Bitte kontaktieren Sie Ihren Administrator.");
54 Assert.assertEquals(cp.getProgram(), 'Z');
55 Assert.assertEquals(cp.getKind(), 'B');
56 Assert.assertEquals(cp.getPriority(), 'D');
57 Assert.assertTrue(cp.getTime() > 0);
58 Assert.assertTrue(cp.getTime() <= System.currentTimeMillis());

```

*Aufräumen der benutzen Komponenten, (@After)*

```
59 IRemoteLoggerCheckPoint cp = _getNextCheckPoint(lastCheckPointENG);
60 Assert.assertEquals(cp.getModule(), 0);
61 Assert.assertEquals(cp.getID(), 1);
62 Assert.assertEquals(cp.getMessage(), "Internal error. Please contact administrator.");
63 Assert.assertEquals(cp.getProgram(), 'Z');
64 Assert.assertEquals(cp.getKind(), 'B');
65 Assert.assertEquals(cp.getPriority(), 'D');
66 Assert.assertTrue(cp.getTime() > 0);
67 Assert.assertTrue(cp.getTime() <= System.currentTimeMillis());
```

Abbildung 19: Überprüfen der CheckPoint-Inhalte

Am Ende muss der Logger noch aufgeräumt werden, da sonst in manchen Fällen der Java-Socket nicht beendet wird und der Port (7733) blockiert bleiben würde.

## 7 Fazit

Der produktive Einsatz des Remote-Loggers wird weitere Anforderungen der Administratoren und ADITO4-Projektentwickler aufzeigen. Es wurde hierdurch eine Möglichkeit geschaffen, direkt auf die Ausgaben des ADITO4-Servers zuzugreifen. Das hat den Vorteil, dass nun nicht mehr per Fernzugriff auf das Hostsystem der ADITO4-Kundenserver verbunden werden muss, um dessen Meldungen zu lesen.

Der Remote-Logger bietet auch im Vergleich zum bisherigen „FileLogger“ den entscheidenden Echtzeit-Vorteil, denn der ADITO4-FileLogger schreibt alle erhaltenen CheckPoints blockweise in seine Datei. Somit werden Ausgaben verzögert geschrieben und es kann erst verspätet auf diese reagiert werden.

Es ist denkbar, dass das Feature des Remote-Loggers noch mit einer Exportfunktion erweitert wird. Somit könnte man Log-Dateien erstellen, die man wiederum mit dem „LogFileViewer“ des ADITO4-Managers betrachten kann.

Ebenso wäre es möglich einen Filter zu implementieren, der alle Nachrichten die der Benutzer nicht sehen möchte, herausfiltert. Beispielsweise werden dann nur noch Nachrichten mit der Priorität „hoch“ angezeigt. Einstellbar soll dies mit verschiedenen Buttons und Eingabefelder werden. Ein Filter nach angemeldeten Benutzern ist von der ADITO-Geschäftsleitung ebenfalls gewünscht, denn somit könnten auftretende Fehler am ADITO4-Client leichter identifiziert und behoben werden.

Eine zusätzliche Erweiterung des Remote-Loggers könnte die Verschlüsselung des Datenaustausches zwischen Remote-Logger-Server und Remote-Logger-Client sein. Dann könnte nahezu komplett ausgeschlossen werden, dass unberechtigte Dritte Zugriff zu den vom Remote-Logger-Server gesendeten Daten erhalten. Hierzu käme SSL in Frage. SSL wurde bereits bei der Kommunikation zwischen ADITO4-Server und ADITO4-Client verwendet, was ein Wiederverwenden von bereits bestehendem Code erlaubt.





## 8 Glossar

CheckPoint	Ein CheckPoint kapselt entweder eine Informationsmeldung oder eine Fehlermeldung der ADITO-Softwareprodukte. Diese besteht aus einer Nachricht und mehreren IDs für Programm, Priorität und Art/Ursache (siehe ??)
Consumer	Das Java-spezifische Interface „Consumer“ repräsentiert eine Operation, die ein einzelnes Argument annimmt und kein Ergebnis zurückgibt
CRM / xRM	Customer Relationship Management / Any Relationship Management Steht für Kundenpflege/-bindung, Datensammlung, Datenpflege, Datenverwaltung und das Ziel, Kundenpotenziale optimal auszuschöpfen
Fassade (Facade)	Eine Fassade bietet eine einheitliche Schnittstelle zu einer Menge von Schnittstellen eines Subsystems. Vereinfacht die Benutzung des Subsystems.
Java-Network-API	Leicht benutzbare Netzwerk-API der Programmiersprache Java. Diese erlaubt es auf bestimmte Netzwerkadressen des Computers zu hören und Nachrichten über das Netzwerk zu senden
Logging	Unter Logging versteht man das Speichern von Prozessen oder Datenänderungen. Diese werden in sogenannten Logdateien hinterlegt bzw. gespeichert. Dies wird in Java meist mit Hilfe der modularen Log4J-API abgebildet.
Logdatei	Eine Logdatei enthält das automatisch geführte Protokoll bestimmter Aktionen von Prozessen auf einem Computersystem.
Remote	Remote bedeutet entfernt, wobei die Entfernung sich darauf bezieht, dass der Benutzer keinen unmittelbaren Kontakt mit dem Remote-Gerät hat.
serialisierbares Objekt	Bezeichnet ein Objekt, das in einen Datenstrom umgewandelt werden und somit über das Netzwerk gesendet werden kann.





## 9 Abbildungsverzeichnis

1	Wasserfallmodell . . . . .	3
2	Grobe Zeitplanung . . . . .	3
3	Bestehender Editor . . . . .	4
4	Model View Presenter vs. Model View Controller . . . . .	5
5	Designentwurf des UX-Designers . . . . .	6
6	Baumstruktur für Entitäten und Funktionen . . . . .	7
7	Datenmodell für Entitäten und Funktionen . . . . .	7
8	Baumstruktur für Browsertestergebnisse . . . . .	8
9	Datenmodell für Browsertestergebnisse . . . . .	8
10	Datenstore . . . . .	8
11	CheckItemButton . . . . .	10
12	CheckItemContainer . . . . .	10
13	Initialisierung des Remote-Logger-Servers . . . . .	11
14	Validieren des Logins . . . . .	11
15	Verbindungsaufbau des Remote-Logger-Clients zum Remote-Logger-Server . . . . .	11
16	Setzen eines empfangenen CheckPoints; Benachrichtigung der wartenden Threads . .	12
17	CheckPoint senden . . . . .	12
18	Warten auf neue CheckPoints; Auslesen von CheckPoints . . . . .	12
19	Überprüfen der CheckPoint-Inhalte . . . . .	13
20	Use-Case-Diagramm . . . . .	16

## 10 Literaturverzeichnis

- Riehle, Dirk (1996): „Entwurfsmuster“

## 11 Anhang

### 11.1 Anwendungsfälle

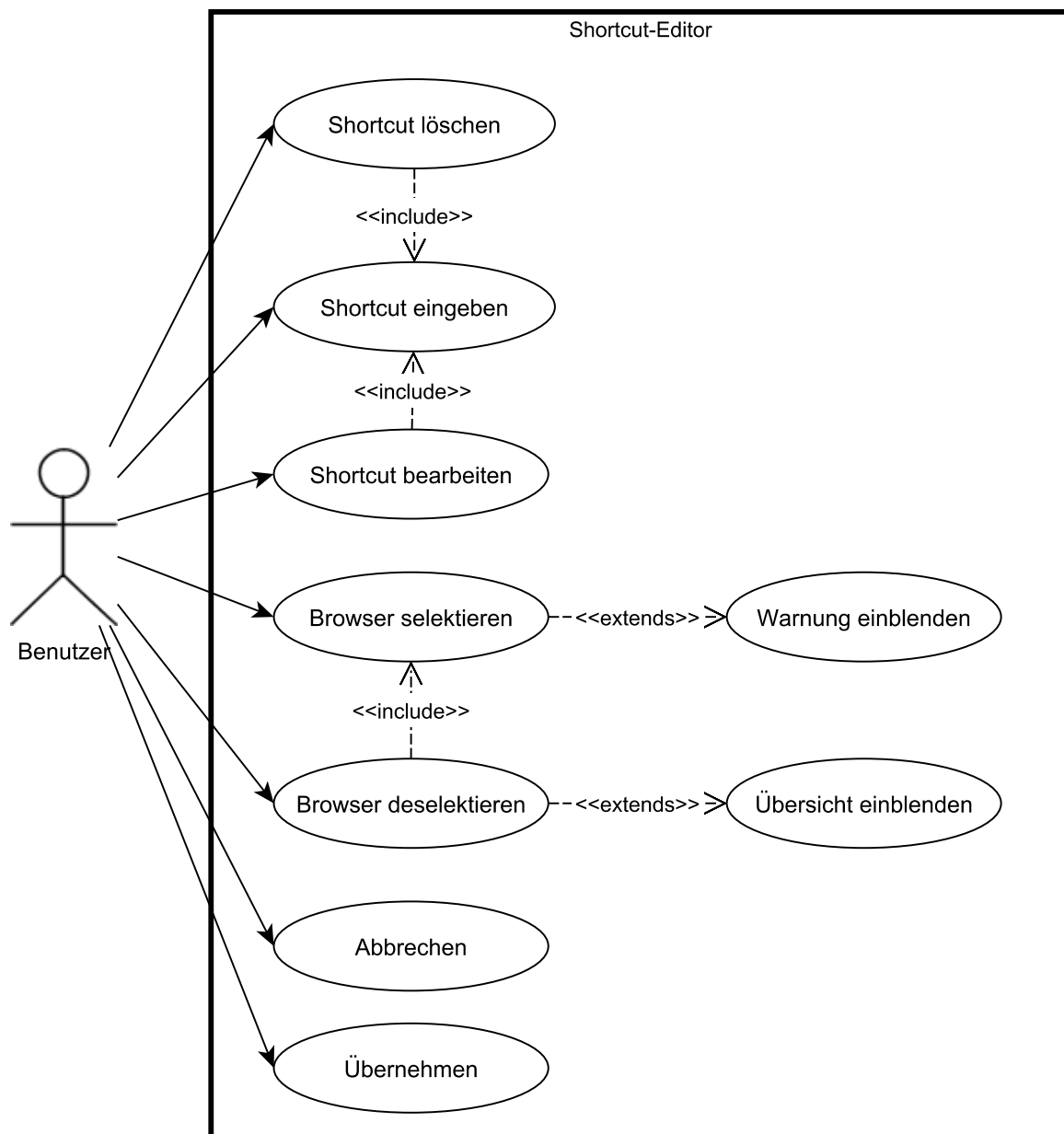


Abbildung 20: Use-Case-Diagramm



## 11.2 Verwendete Ressourcen

### Hardware

- Büroarbeitsplatz (PC, Tastatur, Maus etc.)

### Software

- Windows 10 Professional – Betriebssystem
- IntelliJ IDEA – Entwicklungsumgebung Java
- Maven - Buildsystem
- git – Verteilte Versionsverwaltung
- MiKTeX – Distribution des Textsatzsystems  $\text{\LaTeX}$
- TeXstudio – Entwicklungsumgebung für  $\text{\TeX}$
- yEd Graph Editor - Anwendung zur Erstellung von UML-Diagramme

### Personal

- Mitarbeiter der UX-Abteilung – Erstellung von Designentwürfen
- Entwickler – Umsetzung des Projektes