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

Inhaltsverzeichnis

1 Einleitung

Abbildung 1: Firmengbäude

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.

Die Mitarbeiterzahl des Unternehmens ist insbesondere in den letzten Jahren stark gewachsen. Im Jahr 2018 hat ADITO die Grenze von 100 Mitarbeitern überschritten. Um dafür genügend Raum zu bieten, wurde im gleichen Jahr ein neues Firmengebäude (Abbildung 1) errichtet.

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.

Im Rahmen dieser Arbeit, 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 Entwicklern unserer xRM-Software die Vergabe von Shortcuts zu erleichtern soll ein spezieller Shortcut-Editor erstellt 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 Entwickler feststellen kann, warum ein Shortcut problematisch ist, sollen weitere Informationen angezeigt werden. Diese können beispielsweise angeben, bei welchem Browser bzw. welcher Version das Tastaturkürzel bereits verwendet wird.

1.3 Umfeld

Durchgeführt wird das Projekts in der Entwicklungsabteilung, welche auch für die Umsetzung von ADITO5 zuständig war. Die Notwendigkeit des Editors wurde im Zuge der Weiterentwicklung 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 graphische Benutzeroberfläche dient das Java-Swing-Framework.

1.4 Begründung

Da gewährleistet werden soll, dass vergebene Tastenkürzel auf allen relevanten Browsern funktionieren, muss dem Entwickler 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 verschiedenen Plattformen und testet alle möglichen Shortcut-Kombinationen für die verbreitesten Browser (z.B. Google Chrome, Firefox oder Safari). Das Ergebnis dieser Tests wird in Form von XML-Dateien gespeichert (Beispiel siehe Anhang 11.1). 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 *Propertly* Framework zum Einsatz. Dieses kümmert sich um sämtliche XML-spezifische Arbeiten und ermöglicht so eine konfortable 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 fünf Phasen aufgeteilt (siehe Abbildung 2): Ermittlung der Anforderungen, Erstellung eines Entwurfs, Implementierung, 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.

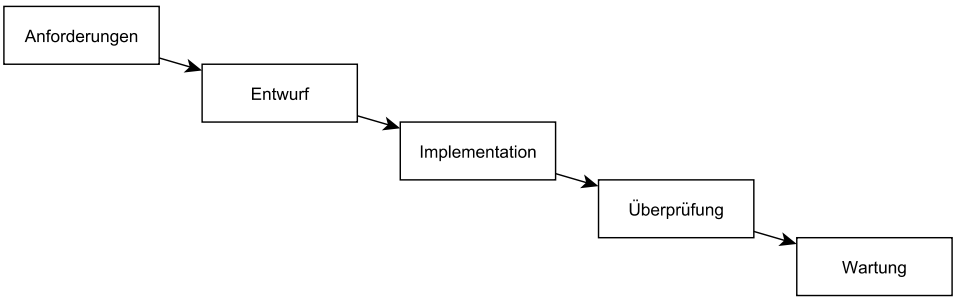


Abbildung 2: Wasserfallmodell

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. Die Zeitplanung lässt sich aus Abbildung 3 entnehmen.

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

Abbildung 3: Grobe Zeitplanung

2.3 Ressourcenplanung

Im Zuge der Ressourcenplanung wurde eine Übersicht (siehe Anhang 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 4). 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.

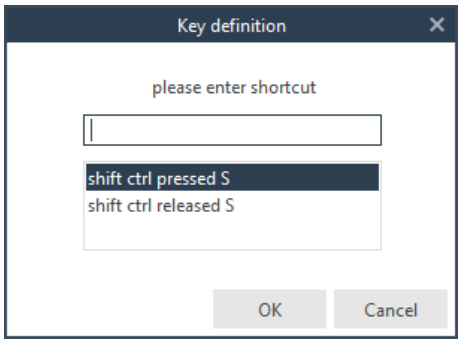


Abbildung 4: Bestehender Editor

Die Aktionen, welchen Shortcuts zugeordnet werden sollen, sind im ADITO Designer schon vorhanden. Zudem existieren sogenannte Entitys, welche Dateneinheiten darstellen und eben genannte Aktionen besitzen können. Beispielsweiße könnte ein Entity „Person“ existieren, welches wiederum die Aktion „Person hinzufügen“ besitzt. Dieser Aktion 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 tabellarischer Form präsentiert. Es soll zu jedem Zeitpunkt ersichtlich sein, für welche Aktion 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 11.3.

3.4 „Make or Buy“-Entscheidung

Am Ende der Analysephase wurde sich die Frage gestellt, ob der Editor selber entwickelt oder gekauft werden soll. Dazu wurde der Markt nach Software durchsucht, welche den Anforderungen dieses Projekts entspricht. Da diese Suche zu keinem Ergebnis gekommen war, wurde sich für die eigene Entwicklung des Editors entschieden.

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 eigendliche 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 5). Der Autor hat sich für das MVP-Entwurfsmuster entschieden, da damit die View auch ohne Model verwendet werden kann und dies für die Zukunft eine höhere Flexibilität ermöglicht.

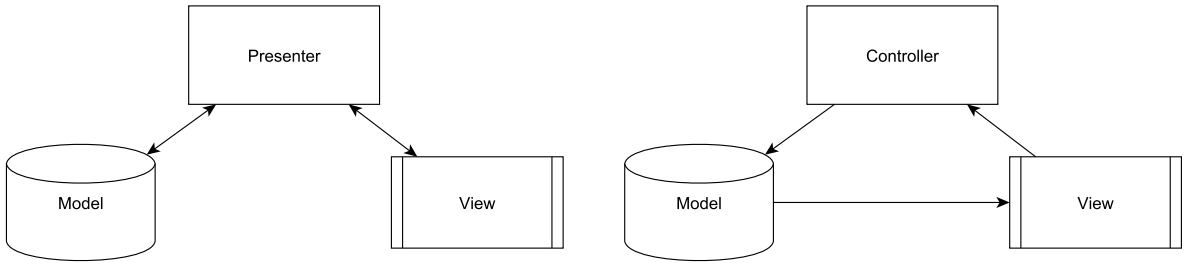


Abbildung 5: Model View Presenter vs. Model View Controller

Bei MVP lässt sich jede Komponente der Software einem der drei Bestandteile - Model, View oder Presenter - zuordnen. Jedes 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 auf dem gleichen Stand. Die lose Kopplung der einzelnen Komponenten erhöht die Wiederverwendbarkeit und Austauschbarkeit. Man könnte beispielsweise die Benutzeroberfläche austauschen, ohne das Model anpassen zu müssen. Für die Verwendung des MVP-Architekturmusters spricht ebenso, dass die einzelnen Komponenten durch die strikte Trennung einfacher getestet, gewartet und flexibel erweitert werden können.

Im Sinne der Wiederverwendbarkeit, werden auch alle GUI-Komponenten des Shortcut Editors völlig seperat 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, kommt zum Lesen der Testergebnis-XML-Dateien das hauseigene *Propertly* Framework zum Einsatz. Dieses Tool stellt Funktionalitäten zum Lesen und Schreiben von XML zur Verfügung. Außerdem kümmert es sich eigenständig 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 6). Im Designentwurf wird unter anderem ersichtlich, welche Komponenten verwendet werden müssen, um alle angeforderten Informationen darzustellen. 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 Aktion der Shortcut gesetzt wird. Zum Anderen ist damit eine intuitive Navigation durch alle Aktionen 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 fokusiertem Zustand erfolgen. Demnach wechselt die Komponente bei Fokusierung 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 Häckchen- oder Kreuzsymbol dar. Somit kann die Komponente visualisieren, bei welchem Browser bzw. Betriebssystem der Shortcut Probleme bereiten kann (Kreuz) und wo dieser unbedenklich ist (Häckchen). Außerdem dient sie dem Benutzer des Editors zur Auswahl eines Elements, um davon mehr Informationen zu erhalten (In der rechten Abbildung des Entwurfs wurden beispielsweise Google Chrome und macOS selektiert).

④ **Shortcut-Tag:** Ein Shortcut-Tag dient zur Darstellung einer Tastenkombination und bietet die Möglichkeit sich mittels eines Kreuz-Buttons selber zu entfernen.

⑤ **TreeTable:** Zur Darstellung der zugrundeliegenden Baumstruktur wird zur Visualisierung aller Aktionen eine Kombination aus Baum 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 detailierte Informationen zu den Testergebnissen bietet. Um nur relevante Daten anzuzeigen, besteht die Möglichkeit einige Sektionen durch Klicken auf den Header einzuklappen.

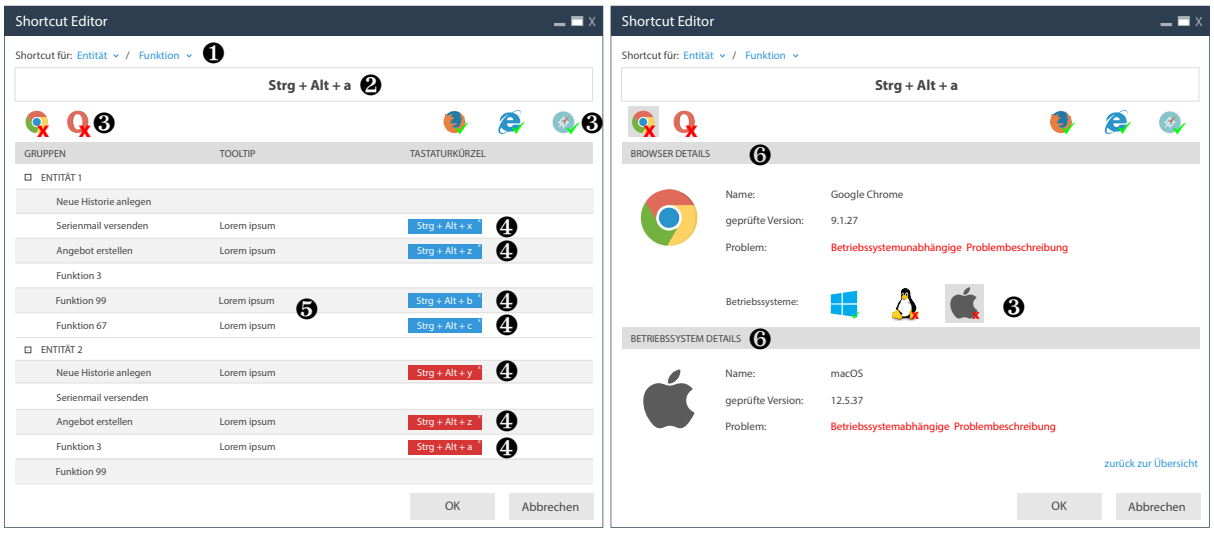


Abbildung 6: Designentwurf des UX-Designers

4.3 Datenmodell für Entitäten und Aktion

Wie sich im Designentwurf (Abbildung 6) – aufgrund der TreeTable – schon erahnen lässt, sollen die Entitäten und deren Aktionen als Baumstruktur aufgebaut werden. Jede Aktion stellt einen Endknoten (Blatt) dar und soll genau einen Shortcut besitzen können. Entitäten hingegen ist es nur erlaubt, Aktionen und weitere Entitäten aufzunehmen (siehe Abbildung 7).

Um diese Struktur im Editor abbilden zu können, wurde ein Datenmodell entworfen, welches den Anforderungen entspricht. Zur Erläuterung des Models ist im Folgenden ein schematisches UML Klassendiagramm abgebildet, welches den Grundaufbau und die Beziehungen zwischen den Elementen verdeutlicht (Abbildung 8).

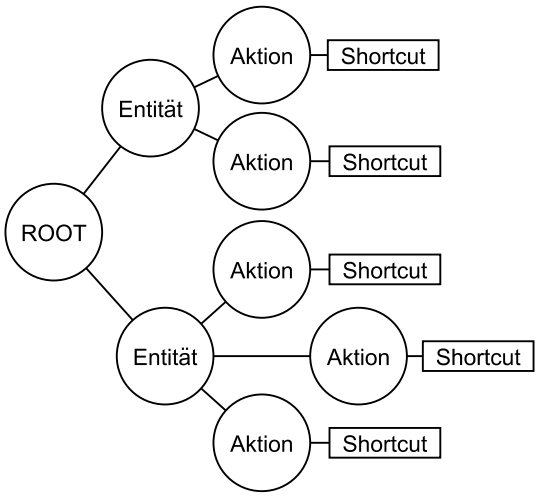


Abbildung 7: Baumstruktur für Entitäten und Funktionen

Zentraler Bestandteil des Datenmodells ist das Interface **IEditorShortcutNode**. Dieses kann neben einem eigenen Namen auch seine Kinder zurückgeben. Diese sind ebenfalls vom Typ **IEditorShortcutNode**. Damit kann grundsätzlich schon eine Baumstrukturen aufgebaut werden. Allerdings ist durch **IEditorShortcutNode** nur die Abbildung der Bestandteile **ROOT** und **Entity** aus Abbildung 7 möglich, da noch kein Tastenkürzel gehalten werden kann.

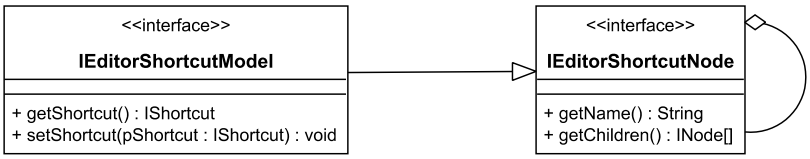


Abbildung 8: Datenmodell für Entitäten und Aktionen

Um auch eine Aktion im Modell abbilden zu können, existiert das Interface **IEditorShortcutModel**. Dieses erbt von **IEditorShortcutNode** und stellt somit einen vollwertigen Knoten dar, welcher bei der Methode **getChildren()** zurückgegeben werden kann. Über die Methoden **setShortcut(...)** kann eine Tastenkombination gesetzt und über **getShortcut()** ausgelesen werden. Da diese Methoden nur die Zuweisung von einem einzigen Shortcut zulassen, ist sichergestellt, dass eine Aktion nur ein Tastenkürzel besitzen kann. Aufgrund der Tatsache, dass **IEditorShortcutModel** 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 konfortabler und effizienter, wenn jeder Knoten die Methode **getChildren()** besitzt, da andernfalls unnötige Abfragen stattfinden müssen.

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

4.4 Datenmodell für Browsertestergebnisse

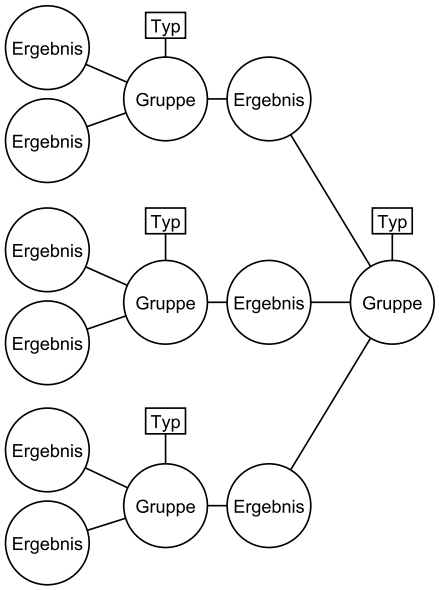


Abbildung 9: Baumstruktur für Browsertestergebnisse

Auch die Browserergebnisse werden als Baumstruktur gehalten. Allerdings ergiebt 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 10 ist ein UML-Klassendiagramm eines Datenmodells abgebildet, welches den Anforderungen zum Halten von Browsertestergebnissen gerecht wird.

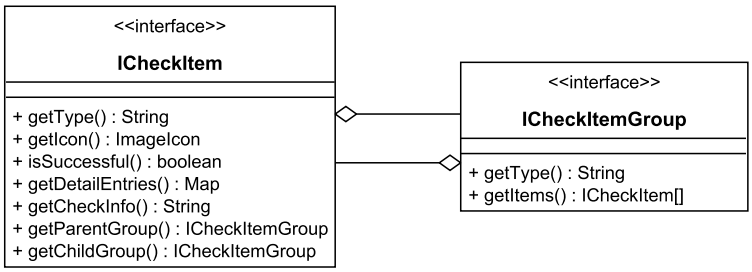


Abbildung 10: Datenmodell für Browsertestergebnisse

In diesem Modell stellt ein **ICheckItem** ein einzelnes Testergbnis 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 durch die Baumstruktur iterieren.

4.5 Datenstore

Damit sowohl die Daten der Entitäten und Aktionen 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.

Um auch ohne die Integration des ShortcutEditors in den ADTIO Designer testweise Entitäten und deren Aktionen zur Verfügung zu haben, soll ein DummyStore implementiert werden, welcher fiktive Testdaten in die Datenmodelle einfügt.

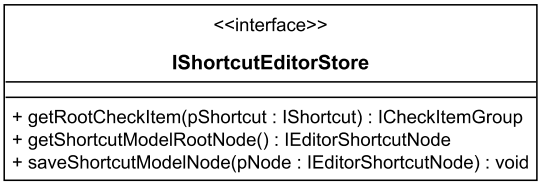


Abbildung 11: Datenstore

5 Implementierung

5.1 GUI Komponenten

Da die Hauptaufgabe dieser Projektarbeit darin bestand, die benötigten GUI Komponenten zu entwickeln und in richtiger Weise zusammenzusetzen, wird darauf im folgenden detailliert eingegangen.

5.1.1 ShortcutField

Ein **ShortcutField** ist – wie in Abschnitt 4.2 bereits erwähnt – für die Anzeige und Bearbeitung von Tastaturkürzeln zuständig. Sofern noch kein Shortcut eingegeben wurde, zeigt sie zudem eine Aufforderung zum Eingeben eines Shortcuts in Form eines Texts an („enter shortcut...“). Um diesen Anforderungen gerecht zu werden, setzt sie sich aus drei bereits bestehenden Komponenten zusammen: Für das Löschen des gesamten Shortcuts dient ein **CrossButton** (umkreistes Kreuz), für die Anzeige und Bearbeitung des Shortcuts kommt die **ShortcutView**-Komponente zum Einsatz und die Textaufforderung wird durch ein **JLabel** realisiert.

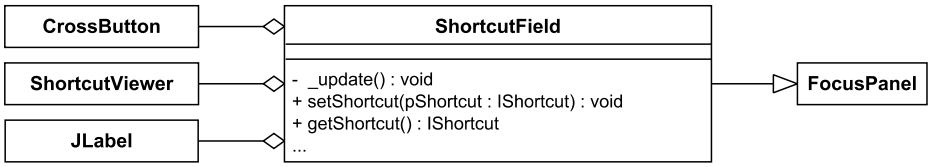


Abbildung 12: ShortcutField

Die eigentliche Funktionalität wird durch die verwendeten Komponenten bereitgestellt. Selbst kümmert sich das **ShortcutField** um die Anzeige der jeweils richtigen Komponente (**JLabel** oder **ShortcutView**), sowie um das Horchen auf Tastatureingaben mittels eines **KeyListeners**. Die private Methode **_update()** ist für das Ein- und Ausblenden der jeweiligen Komponente zuständig (siehe Abbildung 13). Sie wird initial und bei jeder Änderung des Shortcuts aufgerufen.

```
205 private void _update()
206 {
207   boolean empty = getShortcut() == null;
208
209   viewerContainer.setVisible(!empty);
210   textContainer.setVisible(empty);
211
212   closeButton.setVisible(isEditable());
213   closeButton.setEnabled(!empty);
214 }
```

Abbildung 13: Ein- und Ausblenden der Komponenten

Zunächst wird überprüft, ob das **ShortcutField** leer ist (kein Shortcut gesetzt). Das Ergebnis dieser Überprüfung wird in der lokalen Variable **empty** gespeichert. Anschließend werden die Sichtbarkeiten der Komponenten aktualisiert, wobei der **ShortcutViewer** nur bei gesetztem und die Textaufforderung nur bei nicht gesetztem Tastenkürzel angezeigt wird. Der **CrossButton** wird im Editiermodus immer angezeigt aber nur aktiviert, sobald ein Shortcut eingegeben wird.

5.1.2 Check-Button

Um die Grundfunktionalität des im Abschnitt 4.2 erwähnten Check-Buttons zu definieren, wurde zunächst das Interface **ICheckComponent** erstellt. Indessen wird festgelegt, dass ein Check-Button einen Checked-Zustand besitzt, welcher angibt, ob innerhalb der Komponente ein grüner Haken (**true**) oder ein rotes Kreuz (**false**) angezeigt werden soll (siehe Abbildung 14).

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 11.4). Insgesamt stellt ein **CheckToggleButton** eine vollwertige CheckComponent dar, welche direkt verwendet werden könnte.

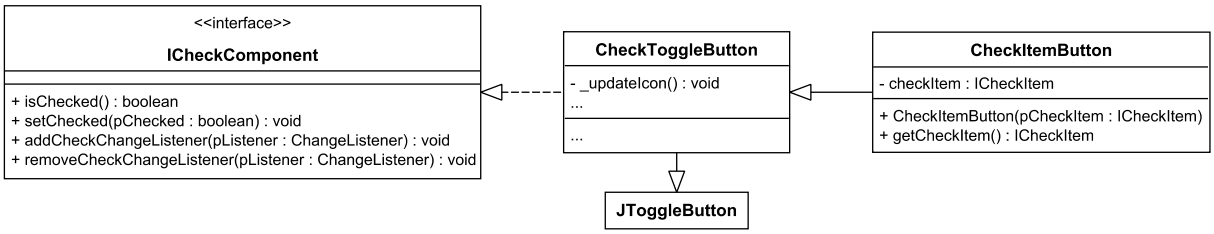


Abbildung 14: CheckItemButton

Um die Benutzung von **CheckToggleButtons** innerhalb des Shortcut Editors bzw. im Zusammenhang mit **ICheckItems** einfacher zu gestalten, wurde die Klasse **CheckItemButton** eingeführt. Diese erweitet **CheckToggleButton** und kann über den Konstruktor ein **ICheckItem** aufnehmen. Wie im Anhang 11.5 ersichtlich, werden innerhalb dieses Konstruktors alle Eigenschaften entsprechend dem **ICheckItem** gesetzt (z.B. Checked-Zustand oder Icon).

5.1.3 CheckItemContainer

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

Um diese Positionierung zu realisieren, wurde der **CheckButtonContainer** erstellt. Dieser stellt eine **JComponent** dar, welcher mittels eines **BoxLayouts** seine Check-Buttons horizontal nebeneinander ausrichtet. Um den zuvor erwähnten Abstand ein- und auszuschalten, existiert die Methode **setSplitting(...)**, welche einen **boolean** erwartet. Ändert sich der Checked-Zustand eines CheckButtons, so kümmert sich der Container auch um die richtige Umsortierung.

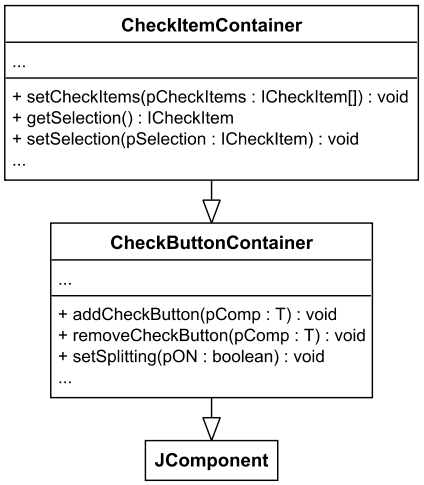


Abbildung 15: CheckItemContainer

Um wiederum die Nutzung innerhalb des Shortcut Editors zu erleichtern, existiert der **CheckItemContainer**. Diesem kann man CheckItems setzten (**setCheckItems(...)**), wonach dieser intern die benötigten Check-Buttons erzeugt und sich selbst hinzufügt oder Unnötige wieder entfernt (siehe Anhang 11.6). Zudem kann über die Methode **getSelection()** der aktuell selektierte Check-Button ausgelesen werden.

5.1.4 CheckItemAccordion

Die Accordion-Komponente ist in der Lage, einen konkreten Pfad des Datenmodells für Browsertestergebnisse mit allen verfügbaren Details darzustellen (siehe Abbildung 6). Jede Sektion des Accordions visualisiert genau ein CheckItem. Sofern Untersektionen vorhanden sind, wird zusätzlich ein CheckItemContainer angezeigt. Dieser dient zur Navigation durch das Datenmodell, wobei immer genau das CheckItem dargestellt wird, welches in der vorherigen Sektion ausgewählt ist.

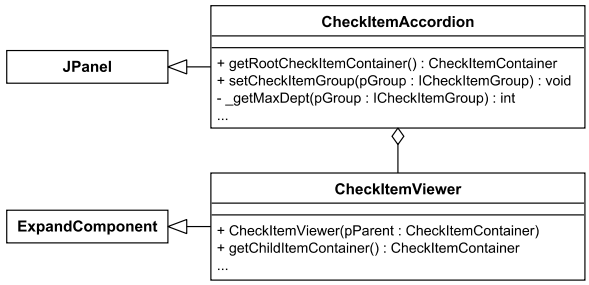


Abbildung 16: CheckItemAccordion

Das **CheckItemAccordion** erbt von der Swing Klasse **JPanel** (siehe Abbildung 16) und verwendet zur vertikalen Positionierung der einzelnen Sektionen das **BoxLayout**. Im Konstruktor wird der oberste **CheckItemContainer** erzeugt, welcher zur Auswahl des Browsers dient und sich nicht innerhalb des Accordions befindet (siehe Abbildung 6). Um auf diesen außerhalb zugreifen zu können, dient die Methode **getRootCheckItemContainer()**.

Eine Sektion wird durch die Klasse **CheckItemViewer** repräsentiert (siehe Abbildung 16). Diese Komponente besitzt die Fähigkeit alle Details eines CheckItems in tabellarischer Form darzustellen. Sie erbt von **ExpandComponent**, wodurch die Funktion des Ein- und Ausklappens und die Anzeige einer Header-Leiste ermöglicht wird. Damit das selektierte **CheckItem** des vorherigen **CheckItemViewers** angezeigt und auf Selektionsänderungen reagiert werden kann, wird im Konstruktor der vorherige **CheckItemContainer** übergeben. Diesem wird ein Listener hinzugefügt, welcher auf Änderungen der Selektion horcht und so die Aktualisierung des Viewers ermöglicht.

In Abbildung 17 ist der Sourcecode der Methode **setCheckItemGroup(...)** abgebildet. In dieser wird anhand der übergebenen **ICheckItemGroup** das Accordion aufgebaut. Zu Beginn wird überprüft, ob es sich bei der übergebenen Gruppe um die bereits gesetzte handelt. Ist dies der Fall, so wird die Methode ohne Änderungen verlassen. Andernfalls werden dem RootCheckItemContainer die CheckItems der RootGruppe (Browser Check-Items) gesetzt. Anschließend wird über die Methode **_getMaxDept(...)** die maximale Tiefe der Baumstruktur von **pRootGroup** ermittelt. Dieses Ergebnis dient als Grundlage für die Anzahl von Sektionen, die eingefügt werden. Nach dem Löschen aller vorhandenen Komponenten, wird in jedem Schleifendurchgang eine Sektion eingefügt. Dabei wird im Konstruktor des **CheckItemViewers** immer der **CheckItemContainer** der vorherigen Sektion oder am Anfang der **rootCheckItemContainer** übergeben. Dadurch wird der richtige Zusammenhang zwischen den Sektionen hergestellt.

```
62 public void setCheckItemGroup(ICheckItemGroup pRootGroup)
63 {
64   if(!Objects.equals(pRootGroup, rootGroup))
65   {
66     rootGroup = pRootGroup;
67
68     rootCheckItemContainer.setCheckItems(pRootGroup.getItems());
69
70     int dept = _getMaxDept(pRootGroup);
71     CheckItemContainer children = rootCheckItemContainer;
72
73     removeAll();
74     for (int i = 0; i < dept; i++)
75     {
76       CheckItemViewer viewer = new CheckItemViewer(children);
77       children = viewer.getChildItemContainer();
78       add(viewer);
79     }
80   }
81 }
```

Abbildung 17: Setzten der CheckItemGroup im CheckItemAccordion

5.1.5 BreadCrumb und TreeTable

Die Komponenten BreadCrumb und TreeTable mussten im Zuge dieser Projektarbeit nicht neu entwickelt werden, da sie bereits zu einem früheren Zeitpunkt umgesetz wurden.

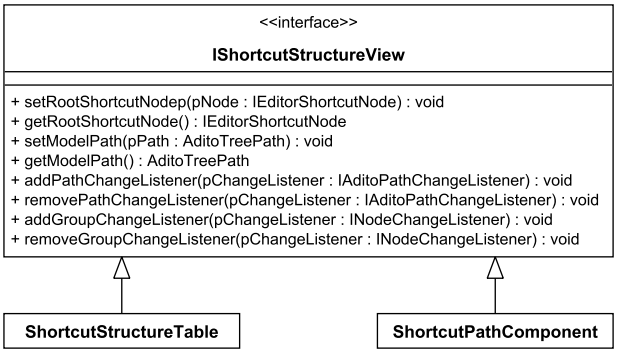


Abbildung 18: IShortcutStructureView

Um sie jedoch im Zusammenhang mit dem MVP-Architekturmuster verwenden zu können, mussten sie in eigene Klassen gekapselt werden, welche **IShortcutStructureView** (siehe Abbildung 18) implementieren. Dazu wurden die Klassen **ShortcutPathComponent** (BreadCrumb) und **ShortcutStructureTable** (TreeTable) erstellt (siehe Anhang 11.7 und 11.8). Über die im Interface enthaltenen Methoden erhält die jeweilige Komponente ihre Informationen, welche zur Darstellung benötigt werden. Außerdem erhält sie die Möglichkeit Wertänderungen über Listener zu kommunizieren. Wählt der Benutzer beispielsweise einen anderen Knoten in der BreadCrumb aus, so werden alle hinzugefügten **PathChangeListener** informiert.

5.1.6 ShortcutEditorUI

Nach der Implementierung der einzelenen Komponenten, mussten diese zu einem gesamten User Interface zusammengesetzt werden. Hierzu wurde die Klasse **ShortcutEditorUI** (siehe Anhang 11.9) erstellt. Sie implementiert das Interface **IShortcutEditorUI**, welches wiederum von **IShortcutStructureView** erbt. Zudem erbt die UI von der Swing Klasse **JPanel** (Abbildung 19).

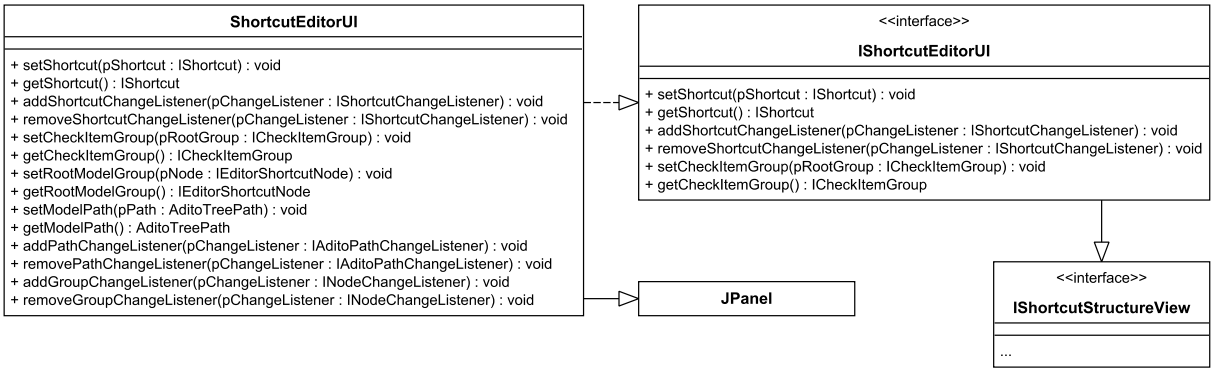


Abbildung 19: ShortcutEditorUI

Zur Anordnung der einzelnen Komponenten kommt das **TableLayout** zum Einsatz. Im Konstruktor der Klasse werden die Spalten- und Zeilenausdehnungen des Layouts definiert und die Komponenten in die entsprechenden Zellen eingefügt. Außerdem wird das **ShortcutStructureModel** und der **ShortcutStructurePresenter** initialisiert.

Da die Funktionalität in den für die UI verwendeten Komponenten enthalten ist, werden alle Getter- und Setter-Anfragen sowie die Listener direkt an das entsprechende Element (Komponente oder Model) weitergeleitet. In der **setShortcut(...)** Methode der ShortcutEditorUI wird beispielsweise sofort die **setShortcut(...)** Methode des ShortcutFields aufgerufen und so der zu setzende Shortcut direkt weitergegeben.

5.2 Presenter

Folgend wird die Implementierung der im Abschnitt 4.1 beschiebenen Presenter erläutert.

5.2.1 ShortcutEditorPresenter

Um die Daten der **ShortcutEditorUI** mit den in Abschnitten 4.3 und 4.4 beschriebenen Datenmodellen abzugleichen, existiert gemäß des MVP-Architekturmusters der **ShortcutEditorPresenter** (siehe Anhang 11.10). Diesem wird über den Konstruktor sowohl der Datenstore als auch eine **IShortcutEditorUI** übergeben. Anschließend werden der UI Listener hinzugefügt, welche auf Shortcut- und Pfadänderungen horchen.

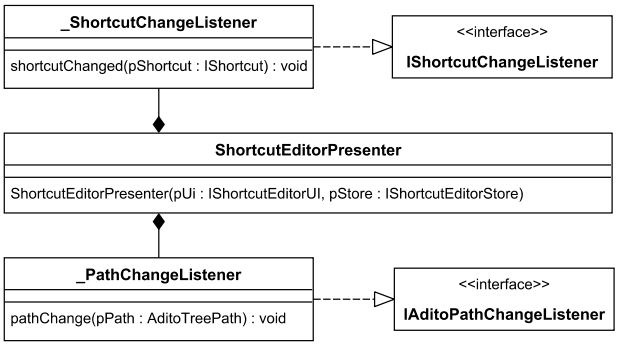


Abbildung 20: ShortcutEditorPresenter

Ändert der Benutzer über die UI den Shortcut, so wird dieser in das entsprechende Datenmodell übertragen. Wird durch den Benutzer der Pfad geändert, also eine andere Aktion selektiert, so lädt der Presenter das Datenmodell der neuen Aktion. Zukünftige Shortcutänderungen werden dann in dieses Datenmodell übertragen.

5.2.2 ShortcutStructurePresenter

Da bei den beiden Komponenten TreeTable und BreadCrumb die selben Daten präsentiert werden, kommt zur Abgleichung ebenfalls das MVP-Architekturmuster zum Einsatz, wobei die beiden genannten Komponenten die Views darstellen. Der Presenter (siehe Anhang 11.11) erhält über den Konstruktor ein **ShortcutStructureModel** und eine beliebige Anzahl von **IShortcutStructureViews**.

In Abbildung 18 wird ersichtlich, dass eine View einen **IEditorShortcutNode** und einen **AditoTreePath** besitzt. Der **IEditorShortcutNode** hält die in Abschnitt 4.3 beschriebene Baumstruktur der Aktionen und der Pfad gibt an, welcher Knoten des Baums gerade selektiert ist. Über Änderung dieser Attribute informieren Listener, welche der View über die entsprechenden Methoden hinzugefügt werden können (z.B. **addPathChangeListener(...)**). Da das **ShortcutStructureModel** die gleiche Funktionalität benötigt, wie die View, implementiert es auch das Interface **IShortcutStructureView** (siehe Anhang 11.12).

Im Konstruktor des **ShortcutStructurePresenters** (siehe Anhang 11.11) werden sowohl dem Model als auch den einzelnen Views Pfad- und Node-Listener hinzugefügt. Löst ein Listener des Models aus, so werden die Änderungen in alle Views übertragen. Löst ein Listener einer View aus, so werden die Änderungen in das Model übertragen. Da dadurch wiederum der Listener des Models anschlägt, werden auch die anderen Views aktualisiert. So wird gewährleistet, dass alle Views und das Model zu jedem Zeitpunkt die gleichen Informationen besitzen.

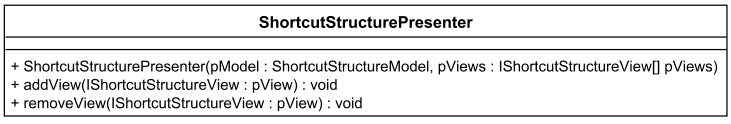


Abbildung 21: ShortcutStructurePresenter

6 Testphase

Um sicherzustellen, das alle implementierten Komponenten anforderungsgerecht funktionieren, wurde der Editor anhand manueller Akzeptanztests getestet. Dabei wurde darauf geachtet, auf jede mögliche Bedienweise einzugehen. So wurde der Editor sowohl per Maus als auch per Tastatur bedient. Gleichzeitig wurde stets überprüft, ob sich der Editor wie gewünscht verhält. Konnte man dabei Unstimmigkeiten feststellen, so wurden die jeweiligen Fehler im Sourcecode behoben und erneut getestet.

Damit die grundlegende Funktionalität auch dauerhaft gewährleistet werden kann, wurde zusätzlich ein automatisierter Test entwickelt (siehe Anhang 11.13). Dabei wurde das Test-Framework *AssertJ Swing* in Kombination mit *JUnit* verwendet.

Das Framework *AssertJ Swing* ist eine Weiterentwicklung des älteren open-source Frameworks FEST. Es ermöglicht die einfache Simulation eines Benutzers für Swing Applikationen. Da es als fluent Api entwickelt wurde, ist der Code sehr gut lesbar und einfach zu verstehen.

Bei *JUnit* handelt es sich ebenfalls um ein quelloffenes Framework, welches durch ein Maven-Plugin in den Build-Vorgang von ADITO eigebunden ist. Dadurch werden alle vorhandenen Tests bei jedem Kompiliervorgang mit Maven automatisch ausgeführt. So können etwaige Fehler schon während der Entwicklung erkannt und vermieden werden.

In Abbildung 22 ist der Sourcode der Initialisieungsmethode **onSetUp()** dargestellt. Um JUnit anzuweisen, die Methode vor jedem Teststart auszuführen, wurde sie mit der Annotaion **@Before** versehen.

```
28 @Before
29 public void onSetUp()
30 {
31   _setLookAndFeel();
32
33   dummyStore = new ShortcutDummyStore();
34   ui = GuiActionRunner.execute(ShortcutEditorUI::new);
35   Dialog shortcutDialog = GuiActionRunner.execute(
36       () -> ShortcutEditorDialog.createDialog(dummyStore, ui));
37
38   dialog = new DialogFixture(shortcutDialog);
39   dialog.show();
40 }
```

Abbildung 22: Setup-Methode

Nach dem Setzen des ADITO spezifischen LookAndFeels, wird der DummyStore und die UI initialisiert. Zum Erstellen des Dialogs werden diese Bestandteile der Methode **ShortcutEditorDialog.createDialog(...)** übergeben. Damit die Ausführung innerhalb des Event Dispatch Threads von Swing erfolgt, kommt ein **GuiActionRunner** zum Einsatz. Dieser führt eine beliebige Anweisung im EDT aus, und liefert das Ergebnis zurück. Nachdem der Dialog erzeugt wurde, wird ein **DialogFixture** erstellt. Hierüber lässt sich der Editor automatisiert bedienen und kann so getestet werden.

```
28 @Test
29 public void test_EnterShortcut()
30 {
31   int i = 0;
32
33   for (IEditorShortcutModel model : dummyStore.getAllModels())
34   {
35     IShortcut shortcut = new Shortcut(EKeyModifier.SHIFT,
36                                       EKey.values()[i++]);
37
38     _selectNode(model);
39     _enterShortcut(shortcut);
40     Assert.assertEquals(model.getShortcut(), shortcut);
41   }
42 }
```

Abbildung 23: Setup-Methode

Die mit **@Test** annotierte Testmethode (Abbildung 22) überprüft, ob eingegebene Shortcuts auch in das selektierte Model übertragen werden. Hierfür wird jedes verfügbare Shortcut Model selektiert und die Eingabe eines Shortcuts simuliert. Der Test ist dann erfolgreich, wenn der Shortcut jedes Models mit dem jeweils zuvor Eingegebenen übereinstimmt.

7 Fazit

Rückwirkend betrachtet, kann festgehalten werden, dass alle Anforderungen an den Editor gemäß der Projektplanung umgesetzt wurden. Der im Abschnitt 2.2 erwähnte Projektplan konnte eingehalten werden. Die Tabelle in Abbildung 24 zeigt die tatsächlich benötigten Zeit gegenüber der geplanten. Dabei ist zu erkennen, dass es in den einzelenen Phasen nur zu geringen Zeitabweichungen gekommen ist. Die entstandene Differenzen haben sich untereinander ausgeglichen, wodurch der veranschlagte Zeitrahmen von 70h genau eingehalten werden konnte.

Vorgang	Soll	Ist	Differenz
1. Analysephase	3 h	3 h	0 h
2. Entwurfsphase	15 h	16 h	+ 1 h
3. Implementierungsphase	40 h	39,5 h	– 0,5 h
4. Testphase	2 h	1,5 h	– 0,5 h
5. Dokumentationserstellung	10 h	10 h	0 h
	70 h	70 h	0 h

Abbildung 24: Soll-/Ist-Vergleich

Auch das Design konnte wie geplant umgesetzt werden. Vergleicht man den fertigen Editor (Abbildung 25) mit dem anfänglich erstellten Designentwurf (Abbildung 6) so erkennt man, dass die Anordnung sowie das Aussehen weitestgehend identisch sind.

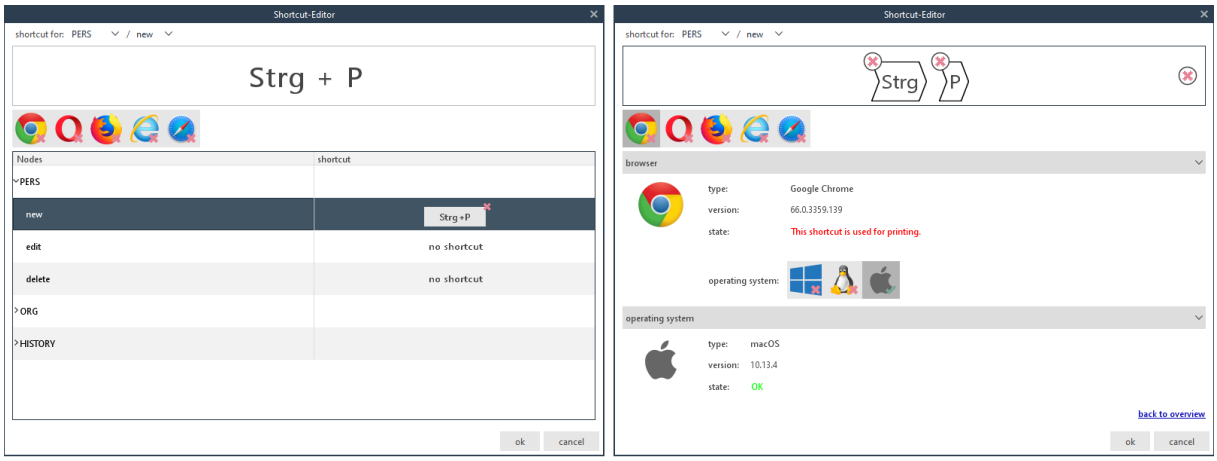


Abbildung 25: Screenshot des Editors

Da der Editor bisher noch nicht in den ADITO Designer integriert ist, sondern in einer eigenen Umgebung gestartet wird, besteht die zukünftige Aufgabe darin, den Editor an den richtigen Stellen des ADITO Designers einzubinden und die Schnittstellen zu entwickeln. Zudem ist es denkbar, dass in Zukunft zusätzliche Anforderungen an den Editor gestellt werden. Beispielsweise könnte es in Zukunft nötig sein, nach gesetzten Shortcuts suchen zu können.

8 Glossar

CRM / xRM	Customer Relationship Management / Any Relationship Management Steht für Kundenpflege/-bindung, Datensammlung, Datenpflege, Datenverwaltung und das Ziel, Kundenpotenziale optimal auszuschöpfen

9 Abbildungsverzeichnis

10 Literaturverzeichnis

- Riehle, Dirk (1996): „Entwurfsmuster“

11 Anhang

11.1 XML

```xml
<?xml version="1.0" encoding="UTF-8"?><ShortcutTestResultModel name="chrome-WINDOWS">
  <browserName>chrome</browserName>
  <browserVersion>66.0.3359.170</browserVersion>
  <osName>Windows 10</osName>
  <osVersion>10.0</osVersion>
  <failedShortcuts>
    <ShortcutTestEntryModel name="e5e6edfc-6f2c-4762-8242-3887f61699c4">
      <shortcut>
        <keyModifiers>
          <KeyModifierDataModel name="SHIFT">
            <keyModifier>SHIFT</keyModifier>
          </KeyModifierDataModel>
          <KeyModifierDataModel name="ALT">
            <keyModifier>ALT</keyModifier>
          </KeyModifierDataModel>
          <KeyModifierDataModel name="CTRL">
            <keyModifier>CTRL</keyModifier>
          </KeyModifierDataModel>
        </keyModifiers>
        <key>A</key>
      </shortcut>
      <success>true</success>
    </ShortcutTestEntryModel>
    <ShortcutTestEntryModel name="95d83f77-be27-4b1c-b0a8-bd0cfb07c0a4">
      <shortcut>
        <keyModifiers>
          <KeyModifierDataModel name="SHIFT">
            <keyModifier>SHIFT</keyModifier>
          </KeyModifierDataModel>
          <KeyModifierDataModel name="META">
            <keyModifier>META</keyModifier>
          </KeyModifierDataModel>
        </keyModifiers>
        <key>INSERT</key>
      </shortcut>
      <success>true</success>
    </ShortcutTestEntryModel>
    <ShortcutTestEntryModel name="cccd89e4-02bd-4638-9220-6d7e2bc9815a">
      <shortcut>
        <keyModifiers>
          <KeyModifierDataModel name="ALT_GRAPH">
            <keyModifier>ALT_GRAPH</keyModifier>
          </KeyModifierDataModel>
          <KeyModifierDataModel name="ALT">
            <keyModifier>ALT</keyModifier>
          </KeyModifierDataModel>
          <KeyModifierDataModel name="CTRL">
            <keyModifier>CTRL</keyModifier>
          </KeyModifierDataModel>
        </keyModifiers>
        <key>NUM0</key>
      </shortcut>
      <success>true</success>
    </ShortcutTestEntryModel>
    <ShortcutTestEntryModel name="3afd18fa-8f23-46aa-bd80-ca3a331fa677">
      <shortcut>
        <keyModifiers>
          <KeyModifierDataModel name="ALT_GRAPH">
            <keyModifier>ALT_GRAPH</keyModifier>
          </KeyModifierDataModel>
          <KeyModifierDataModel name="SHIFT">
            <keyModifier>SHIFT</keyModifier>
          </KeyModifierDataModel>
          <KeyModifierDataModel name="CTRL">
            <keyModifier>CTRL</keyModifier>
          </KeyModifierDataModel>
          <KeyModifierDataModel name="META">
            <keyModifier>META</keyModifier>
          </KeyModifierDataModel>
        </keyModifiers>
        <key>F10</key>
      </shortcut>
      <success>true</success>
    </ShortcutTestEntryModel>
    ...
```

11.2 Verwendete Ressourcen

Hardware

- Büroarbeitsplatz (PC, Tastatur, Maus etc.)

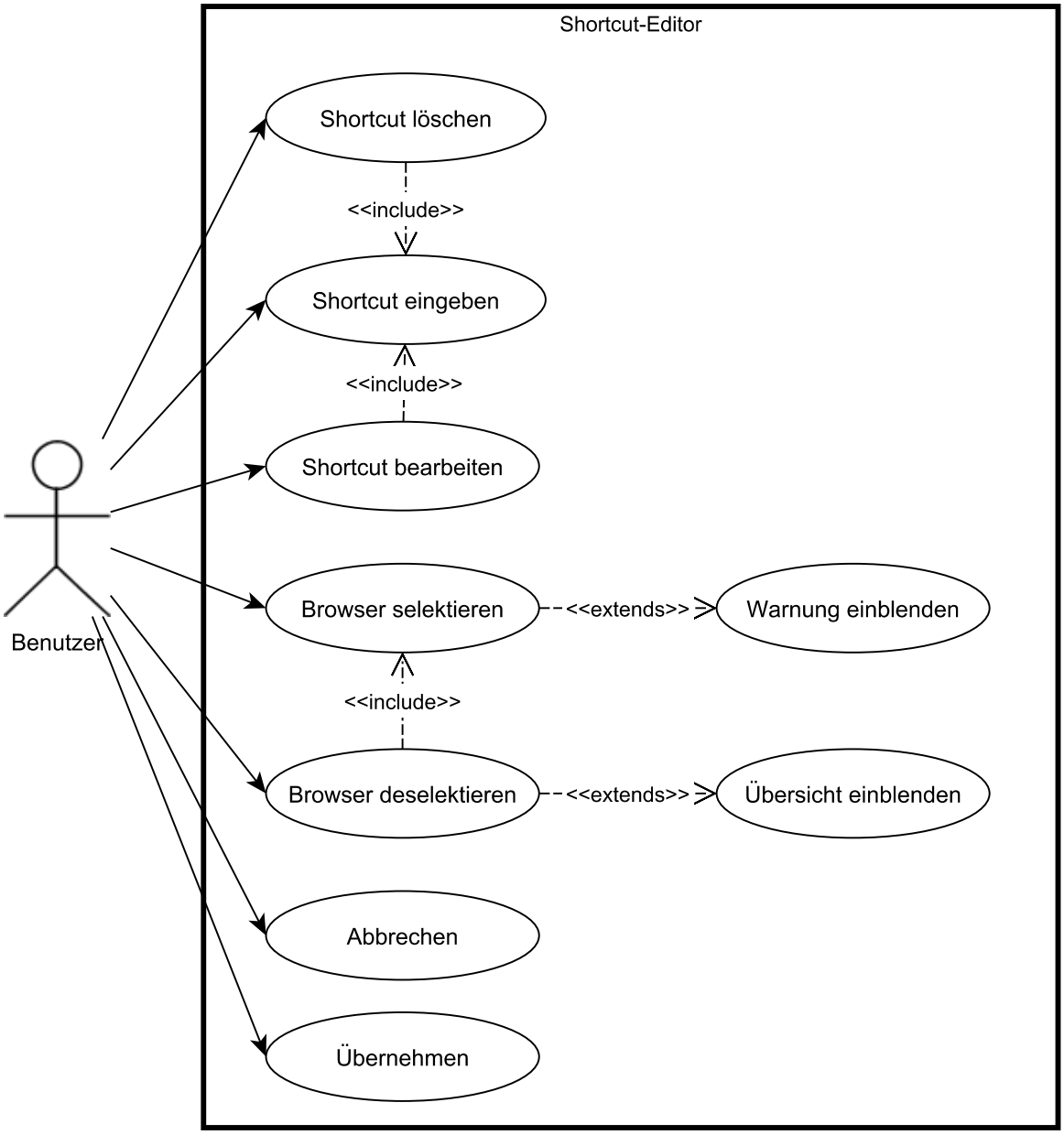
Software

- Windows 10 Professional – Betriebssystem
- IntelliJ IDEA – Entwicklungsumgebung Java
- Java JDK - Software Developer Kit (SDK)
- Maven - Buildsystem
- git – Verteilte Versionsverwaltung
- MiKTeX – Distribution des Textsatzsystems LaTeX
- TeXstudio – Entwicklungsumgebung für TeX
- yEd Graph Editor - Anwendung zur Erstellung von UML-Diagramme

Personal

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

11.3 Anwendungsfälle



11.4 Zeichnen des Icons im CheckToggleButton

```
20   /**
21   * Aktualisiert das Icon
22   */
23   private void _updateIcon()
24   {
25     int width = icon.getWidth(null);
26     int height = icon.getHeight(null);
27
28     Image stateIcon = checked ? checkedIcon : uncheckedIcon;
29
30     int x2 = width - stateIcon.getWidth(null);
31     int y2 = height - stateIcon.getHeight(null);
32
33     BufferedImage image = new BufferedImage(width, height, BufferedImage.TYPE_INT_ARGB);
34
35     Graphics g = image.getGraphics();
36
37     g.drawImage(icon, 0, 0, null);
38     g.drawImage(stateIcon, x2, y2, null);
39
40     setIcon(new ImageIcon(image));
41   }
```

11.5 CheckItemButton

```
12 public class CheckItemButton extends CheckToggleButton
13 {
14   private final ICheckItem checkItem;
15
16   /**
17   * Konstruktor
18   *
19   * @param pCheckItem CheckItem
20   */
21   public CheckItemButton(ICheckItem pCheckItem)
22   {
23     super(pCheckItem.getIcon().getImage());
24
25     checkItem = pCheckItem;
26
27     setChecked(checkItem.isSuccessful());
28     setToolTipText(checkItem.getType());
29     setPreferredSize(IShortcutEditorConstants.CHECK_ITEM_SIZE);
30     setMinimumSize(IShortcutEditorConstants.CHECK_ITEM_SIZE);
31     setMaximumSize(IShortcutEditorConstants.CHECK_ITEM_SIZE);
32     setOpaque(true);
33   }
34
35   /**
36   * Gibt das zugehörige ICheckItem zurück
37   *
38   * @return zugehörige ICheckItem
39   */
40   public ICheckItem getCheckItem()
41   {
42     return checkItem;
43   }
44 }
```

11.6 Setzen der CheckItems im CheckItemContainer

```
43 /**
44 * Setzt die zu visualisierenden ICheckItems
45 *
46 * @param pCheckItems zu visualisierenden ICheckItems
47 */
48 public void setCheckItems(ICheckItem[] pCheckItems)
49 {
50   ICheckItem selection = getSelection();
51
52   for (CheckItemButton comp : getAllCheckButtons())
53   {
54     comp.removeItemListener(itemListener);
55     removeCheckButton(comp);
56   }
57
58   if(pCheckItems != null)
59   {
60     for (ICheckItem item : pCheckItems)
61     {
62       CheckItemButton component = new CheckItemButton(item);
63       component.addItemListener(itemListener);
64       addCheckButton(component);
65     }
66   }
67
68   revalidate();
69   repaint();
70 }
```

11.7 ShortcutStructureTable

```
25 public class ShortcutStructureTable extends JPanel implements IShortcutStructureView
26 {
27   private SplitsTreeTable table;
28   private ShortcutStructureTreeModel treeModel;
29
30   private ShortcutStructureModel model;
31
32   /**
33   * Konstruktor
34   */
35   public ShortcutStructureTable()
36   {
37     setLayout(new BorderLayout());
38
39     model = new ShortcutStructureModel();
40
41     table = new SplitsTreeTable();
42     table.setRootVisible(false);
43     table.setRenderDataProvider(new _ModelProvider());
44     table.setShowVerticalLines(false);
45     table.setShowHorizontalLines(false);
46     table.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
47     table.setRowHeight(IShortcutEditorConstants.TABLE_ROW_HEIGHT);
48
49     add(new JScrollPane(table), BorderLayout.CENTER);
50   }
51
52   @Override
53   public IEditorShortcutNode getRootShortcutNode()
54   {
55     return model.getRootShortcutNode();
56   }
57
58   @Override
59   public void setNodePath(AditoTreePath<IEditorShortcutNode> pPath)
60   {
61     model.setNodePath(pPath);
62   }
63
64 ...
```

11.8 ShortcutPathComponent

```
19 public class ShortcutPathComponent extends JPanel implements IShortcutStructureView
20 {
21   private static final int GAP = 5;
22
23   private BreadCrumb<IEditorShortcutNode> breadCrumb;
24
25   /**
26   * Konstruktor
27   */
28   public ShortcutPathComponent()
29   {
30     double[] x = {TableLayout.PREFERRED, GAP, TableLayout.MINIMUM};
31     double[] y = {TableLayout.PREFERRED};
32
33     setLayout(new TableLayout(x, y));
34
35     TableLayoutUtil tlu = new TableLayoutUtil(this);
36
37     JLabel textLabel = new JLabel(IShortcutEditorConstants.TEXT_SHORTCUT_FOR);
38     tlu.add(0, 0, textLabel);
39
40     breadCrumb = new BreadCrumb<>();
41     tlu.add(2, 0, breadCrumb);
42   }
43
44   @Override
45   public void setRootShortcutNode(IEditorShortcutNode pNode)
46   {
47     breadCrumb.setRootNode(pNode);
48
49     synchronized (nodeChangeListeners)
50     {
51       for (INodeChangeListener listener : nodeChangeListeners)
52       listener.groupChanged(pNode);
53     }
54   }
55
56   @Override
57   public IEditorShortcutNode getRootShortcutNode()
58   {
59     return breadCrumb.getRootNode();
60   }
61
62   @Override
63   public void setNodePath(AditoTreePath<IEditorShortcutNode> pPath)
64   {
65     breadCrumb.setPath(pPath);
66   }
67
68   @Override
69   public AditoTreePath<IEditorShortcutNode> getNodePath()
70   {
71     return breadCrumb.getPath();
72   }
73
74   @Override
75   public void addPathChangeListener(IAditoPathChangeListener<IEditorShortcutNode>
76                                     pChangeListener)
77   {
78     breadCrumb.addPathChangeListener(pChangeListener);
79   }
80
81   @Override
82   public void removePathChangeListener(IAditoPathChangeListener<IEditorShortcutNode>
83                                        pChangeListener)
84   {
85     breadCrumb.removePathChangeListener(pChangeListener);
86   }
87
88 ...
```

11.9 ShortcutEditorUI

```
25 public class ShortcutEditorUI extends JPanel implements IShortcutEditorUI
26 {
27   private ShortcutField shortcutField;
28   private CheckItemContainer checkContainer;
29   private CheckItemAccordion checkAccordion;
30
31   private ShortcutStructureModel model;
32
33   private static final int GAP = 5;
34
35   /**
36   * Konstruktor
37   */
38   public ShortcutEditorUI()
39   {
40     double[] x = {10, TableLayout.FILL, 10};
41     double[] y = {GAP,
42                   20,
43                   GAP,
44                   80,
45                   GAP,
46                   50,
47                   GAP,
48                   TableLayout.FILL,
49                   10};
50
51     setLayout(new TableLayout(x, y));
52
53     TableLayoutUtil tlu = new TableLayoutUtil(this);
54
55     model = new ShortcutStructureModel();
56
57     ShortcutPathComponent pathComp = new ShortcutPathComponent();
58     ShortcutStructureTable modelViewer = new ShortcutStructureTable();
59
60     new ShortcutStructurePresenter(model, pathComp, modelViewer);
61
62     shortcutField = new ShortcutField();
63     checkAccordion = new CheckItemAccordion();
64     checkContainer = checkAccordion.getRootCheckItemContainer();
65
66     tlu.add(1, 1, pathComp);
67     tlu.add(1, 3, shortcutField);
68     tlu.add(1, 5, checkContainer);
69     tlu.add(1, 7, checkAccordion);
70     tlu.add(1, 7, modelViewer);
71   }
72
73   public void setShortcut(IShortcut pShortcut)
74   {
75     shortcutField.setShortcut(pShortcut);
76   }
77
78   public IShortcut getShortcut()
79   {
80     return shortcutField.getShortcut();
81   }
82
83   public void addShortcutChangeListener(IShortcutChangeListener pChangeListener)
84   {
85     shortcutField.addShortcutChangeListener(pChangeListener);
86   }
87
88   public void removeShortcutChangeListener(IShortcutChangeListener pChangeListener)
89   {
90     shortcutField.removeShortcutChangeListener(pChangeListener);
91   }
92
93   public void setCheckItemGroup(ICheckItemGroup pRootGroup)
94   {
95     checkAccordion.setCheckItemGroup(pRootGroup);
96   }
97
98   @Override
99   public ICheckItemGroup getCheckItemGroup()
100  {
101    return checkAccordion.getCheckItemGroup();
102  }
103
104 ...
```

11.10 ShortcutEditorPresenter

```
19 public class ShortcutEditorPresenter
20 {
21   private final IShortcutEditorUI ui;
22   private final IShortcutEditorStore editorStore;
23   private final IShortcutChangeListener shortcutChangeListener;
24   private IEditorShortcutModel currModel;
25
26   /**
27   * @param pUi zu steuernde UI
28   * @param pStore Store zum beziehen der Daten
29   */
30   public ShortcutEditorPresenter(IShortcutEditorUI pUi, IShortcutEditorStore pStore)
31   {
32     ui = pUi;
33     editorStore = pStore;
34
35     shortcutChangeListener = new _ShortcutListener();
36
37     ui.addShortcutChangeListener(shortcutChangeListener);
38     ui.addPathChangeListener(new _PathListener());
39     ui.setRootShortcutNode(pStore.getShortcutModelRootNode());
40   }
41
42   /**
43   * Extrahiert aus dem Pfad das ShortcutModel, auf welchen sich der Pfad bezieht
44   *
45   * @param pPath Pfad zum Model
46   * @return ShortcutModel
47   */
48   private IEditorShortcutModel _getShortcutModel(AditoTreePath<IEditorShortcutNode> pPath)
49   {
50     AditoTreePath<IEditorShortcutNode> node = pPath;
51
52     while (node != null && !(node.getNode() instanceof IEditorShortcutModel))
53       node = node.getNext();
54
55     if(node == null)
56       return null;
57
58     return (IEditorShortcutModel) node.getNode();
59   }
60
61   /**
62   * Listener zum handeln von Pfadänderungen
63   */
64   private class _PathListener implements IAditoPathChangeListener<IEditorShortcutNode>
65   {
66     @Override
67     public void pathChange(AditoTreePath<IEditorShortcutNode> pPath)
68     {
69       if(currModel != null)
70         currModel.removeShortcutChangeListener(shortcutChangeListener);
71
72       currModel = _getShortcutModel(pPath);
73
74       if (currModel != null)
75       {
76         currModel.addShortcutChangeListener(shortcutChangeListener);
77         ui.setShortcut(currModel.getShortcut());
78       }
79     }
80   }
81   /**
82   * Listener zum handeln von Shortcutänderungen
83   */
84   private class _ShortcutListener implements IShortcutChangeListener
85   {
86     @Override
87     public void shortcutChanged(IShortcut pShortcut)
88     {
89       if (!Objects.equals(pShortcut, ui.getShortcut()))
90         ui.setShortcut(pShortcut);
91
92       if (currModel != null && !Objects.equals(pShortcut, currModel.getShortcut()))
93         currModel.setShortcut(pShortcut);
94
95       ICheckItemGroup rootCheckItem = editorStore.getRootCheckItem(pShortcut);
96       ui.setCheckItemGroup(rootCheckItem);
97     }
98   }
99 }
```

11.11 ShortcutStructurePresenter

```
21  public class ShortcutStructurePresenter
22  {
23    private final INodeChangeListener groupChangeListener;
24    private final IAditoPathChangeListener<IEditorShortcutNode> pathChangeListener;
25
26    private final ShortcutStructureModel model;
27    private final List<IShortcutStructureView> views;
28
29    /**
30    * Konstruktor
31    *
32    * @param pModel Das Hauptmodel
33    * @param pViews belibige Anzahl von Views
34    */
35    public ShortcutStructurePresenter(ShortcutStructureModel pModel,
36                                      IShortcutStructureView... pViews)
37    {
38      model = pModel;
39      views = new ArrayList<>();
40
41      groupChangeListener = new _ShortcutNodeChangeListener();
42      pathChangeListener = new _ShortcutPathChangeListener();
43
44      model.addPathChangeListener(this::_updatePath);
45      model.addNodeChangeListener(this::_updateGroup);
46
47      if(pViews != null)
48      for (IShortcutStructureView view : pViews)
49      addView(view);
50    }
51
52    /**
53    * Fügt dem Presenter eine View hinzu
54    *
55    * @param pView hinzuzufügende View
56    */
57    public void addView(IShortcutStructureView pView)
58    {
59      synchronized (views)
60      {
61        views.add(pView);
62      }
63
64      pView.addPathChangeListener(pathChangeListener);
65      pView.addNodeChangeListener(groupChangeListener);
66
67      pView.setRootShortcutNode(model.getRootShortcutNode());
68      pView.setNodePath(model.getNodePath());
69    }
70
71    /**
72    * Entfernt eine View
73    *
74    * @param pView zu entfernende View
75    */
76    public void removeView(IShortcutStructureView pView)
77    {
78      synchronized (views)
79      {
80        views.remove(pView);
81      }
82
83      pView.removePathChangeListener(pathChangeListener);
84      pView.removeNodeChangeListener(groupChangeListener);
85    }
86
87    /**
88    * Aktualisiert die ShortcutModels in den einzelenen Views
89    *
90    * @param pGroup zu sentzende Gruppe mit ShortcutModels
91    */
92    private void _updateGroup(IEditorShortcutNode pGroup)
93    {
94      synchronized (views)
95      {
96        for (IShortcutStructureView view : views)
97        if (!Objects.equals(view.getRootShortcutNode(), pGroup))
98        view.setRootShortcutNode(pGroup);
99      }
100   }
```

```
100   /**
101   * Aktualisiert den Pfad in den einzelen Views
102   *
103   * @param pPath zu setztender Pfad
104   */
105   private void _updatePath(AditoTreePath<IEditorShortcutNode> pPath)
106   {
107     synchronized (views)
108     {
109       for (IShortcutStructureView view : views)
110       if (!Objects.equals(view.getNodePath(), pPath))
111       view.setNodePath(pPath);
112     }
113   }
114
115   /**
116   * Listener, welcher Änderungen der ShortcutGroup in das Hauptmodel überträgt
117   */
118   private class _ShortcutNodeChangeListener implements INodeChangeListener
119   {
120     @Override
121     public void groupChanged(IEditorShortcutNode pNode)
122     {
123       if (!Objects.equals(model.getRootShortcutNode(), pNode))
124       model.setRootShortcutNode(pNode);
125     }
126   }
127
128   /**
129   * Listener, welcher Änderungen der ShortcutPath in das Hauptmodel überträgt
130   */
131   private class _ShortcutPathChangeListener
132                 implements IAditoPathChangeListener<IEditorShortcutNode>
133   {
134     @Override
135     public void pathChange(AditoTreePath<IEditorShortcutNode> pPath)
136     {
137       if (!Objects.equals(model.getNodePath(), pPath))
138       model.setNodePath(pPath);
139     }
140   }
141 }
```

11.12 ShortcutStructureModel

```
21 public class ShortcutStructureModel implements IShortcutStructureView
22 {
23   private final List<IAditoPathChangeListener<IEditorShortcutNode>>
24        pathChangeListeners = new ArrayList<>();
25   private final List<INodeChangeListener> nodeChangeListeners = new ArrayList<>();
26
27   private IEditorShortcutNode root;
28   private AditoTreePath<IEditorShortcutNode> path;
29
30   @Override
31   public void setRootShortcutNode(IEditorShortcutNode pNode)
32   {
33     IEditorShortcutNode oldGroup = root;
34
35     root = pNode;
36
37     if(!Objects.equals(oldGroup, pNode))
38      _fireEditorShortcutGroupChangeEvent();
39   }
40
41   @Override
42   public IEditorShortcutNode getRootShortcutNode() { return root; }
43
44   @Override
45   public void setNodePath(AditoTreePath<IEditorShortcutNode> pPath)
46   {
47     if(!Objects.equals(path, pPath))
48     {
49       path = pPath;
50
51       _fireEditorShortcutPathChangeEvent();
52     }
53   }
54
55   @Override
56   public AditoTreePath<IEditorShortcutNode> getNodePath() { return path; }
57
58   @Override
59   public void addPathChangeListener(IAditoPathChangeListener<IEditorShortcutNode>
60                                       pChangeListener)
61   {
62     synchronized (pathChangeListeners)
63     {
64       pathChangeListeners.add(pChangeListener);
65     }
66   }
67
68   @Override
69   public void addNodeChangeListener(INodeChangeListener pChangeListener)
70   {
71     synchronized (nodeChangeListeners)
72     {
73       nodeChangeListeners.add(pChangeListener);
74     }
75   }
76
77   /**
78   * Feuert ein GroupChangeEvent
79   */
80   private void _fireEditorShortcutGroupChangeEvent()
81   {
82     synchronized (nodeChangeListeners)
83     {
84       for (INodeChangeListener listener : nodeChangeListeners)
85        listener.groupChanged(root);
86     }
87   }
88
89   /**
90   * Feuert ein PathChangeEvent
91   */
92   private void _fireEditorShortcutPathChangeEvent()
93   {
94     synchronized (pathChangeListeners)
95     {
96       for (IAditoPathChangeListener<IEditorShortcutNode> listener : pathChangeListeners)
97        listener.pathChange(path);
98     }
99   }
100
101 ...
```

11.13 ShortcutEditor Test

```
21  public class Test_ShortcutEditor
22  {
23    private DialogFixture dialog;
24    private ShortcutDummyStore dummyStore;
25    private ShortcutEditorUI ui;
26
27    @Before
28    public void onSetUp()
29    {
30      _setLookAndFeel();
31
32      dummyStore = new ShortcutDummyStore();
33      ui = GuiActionRunner.execute(ShortcutEditorUI::new);
34      Dialog shortcutDialog = GuiActionRunner.execute(
35      () -> ShortcutEditorDialog.createDialog(dummyStore, ui));
36
37      dialog = new DialogFixture(shortcutDialog);
38      dialog.show();
39    }
40
41    @Test
42    public void test_EnterShortcut()
43    {
44      int i = 26;
45
46      for (IEditorShortcutModel model : dummyStore.getAllModels())
47      {
48        IShortcut shortcut = new ExtendedShortcut(EKeyModifier.SHIFT, EKey.values()[i++]);
49
50        _selectNode(model);
51
52        _enterShortcut(shortcut);
53
54        Assert.assertEquals(model.getShortcut(), shortcut);
55      }
56    }
57
58    /**
59    * Selektiert den übergebenen Node
60    *
61    * @param pNode zu selektierender Node
62    */
63    private void _selectNode(IEditorShortcutNode pNode)
64    {
65      GuiActionRunner.execute(() -> ui.setNodePath(pNode.getPath()));
66      Assert.assertEquals(dialog.table().selectionValue(), pNode.toString());
67    }
68
69    /**
70    * Simuliert die Eingabe eines Shortcuts
71    *
72    * @param pShortcut zu simulierender Shortcut
73    */
74    private void _enterShortcut(IShortcut pShortcut)
75    {
76      JPanelFixture shortcutField = dialog.panel(SHORTCUT_FIELD_NAME).focus();
77
78      SwingShortcut swingShortcut = new SwingShortcut(pShortcut);
79
80      List<Integer> keys = swingShortcut.getAllDefaultSpecKeys();
81
82      for (Integer i : keys)
83      shortcutField.pressKey(i);
84
85      for (Integer i : keys)
86      shortcutField.releaseKey(i);
87
88      Assert.assertEquals(ui.getShortcut(), pShortcut);
89    }
90
91    private void _setLookAndFeel()
92    {
93      try
94      {
95        LookAndFeelSetter.set();
96      }
97      catch (Exception e)
98      {
99        throw new RuntimeException(e);
100     }
101   }
102 }
```