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

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 Unternhemen 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 eigendliche 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 eingebene 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 Entwickung, welche auch für die Umsetzung von ADITO5 zuständig war. Im Zuge der Weiterentwicklung wurde innerhab 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 verbreitesten 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 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 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.

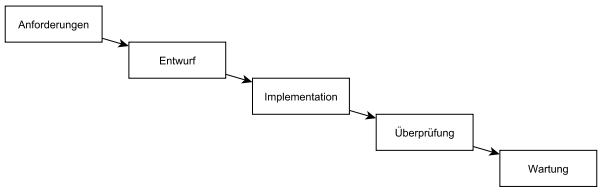


Abbildung 1: Wasserfallmodell

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. Die grobe Zeitplanung der Hauptphasen lässt sich aus Abbildung 2 entnehmen. Eine detailierte zeitliche Planung, bei welcher jede Phase in ihre Unterphasen zerlegt wurde, befindet sich im Anhang (XXX)

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.

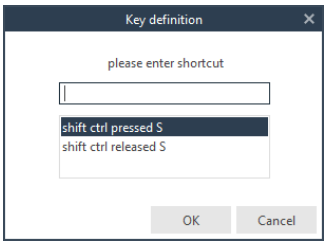


Abbildung 3: Bestehender Editor

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 40 auf Seite 23.

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 Anwendugsfällen 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 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 4). Der Autor hat sich für das MVP-Entwurfsmuster entschieden, da damit die View auch ohne Model verwendet werden kann und es denkbar ist, das diese Möglichkeit in Zukunft benötigt wird.

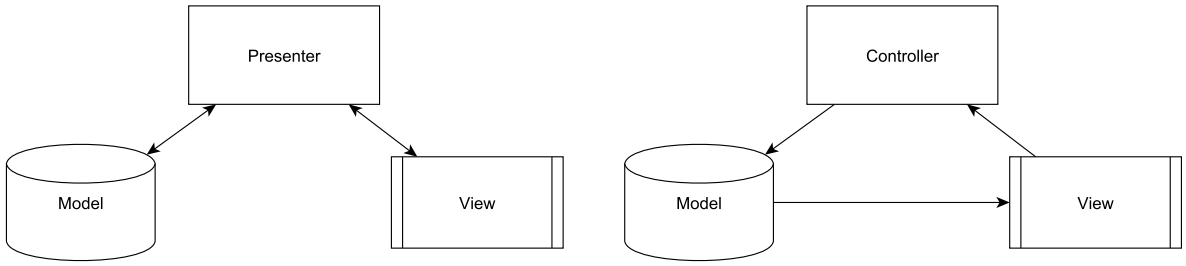


Abbildung 4: Model View Presenter vs. Model View Controller

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 beispielsweiße 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 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 wird für das Lesen der Testergebnis XML-Dateien das hauseigene Propertly 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 angeordet 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 selektiertem Zustand erfolgen. Demnach wechselt die Komponente bei Selektion 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 X-Symbol dar. Somit kann die Komponente visualisieren, bei welchem Browser bzw. Betriebssystem der Shortcut Probleme bereiten kann (X) und wo dieser unbedenklich ist (Häckchen). Außerdem dient sie dem Benutzer zur Auswahl eines Elements, um davon mehr Informationen zu erhalten (Im Entwurf werden beispielhaft für Google Chrome und macOS detailierte Informationen angezeigt).

④ **Shortcut-Tag:** Ein Shortcut-Tag dient zur Darstellung einer Tastenkombination und bietet die Möglichkeit sich mittles 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 detailierte Informationen zu den Testergebnissen bietet. Um nur relevante Daten anzuzeigen, besteht die Möglichkeit einige Sektionen durch Klicken auf den Header einzuklappen.

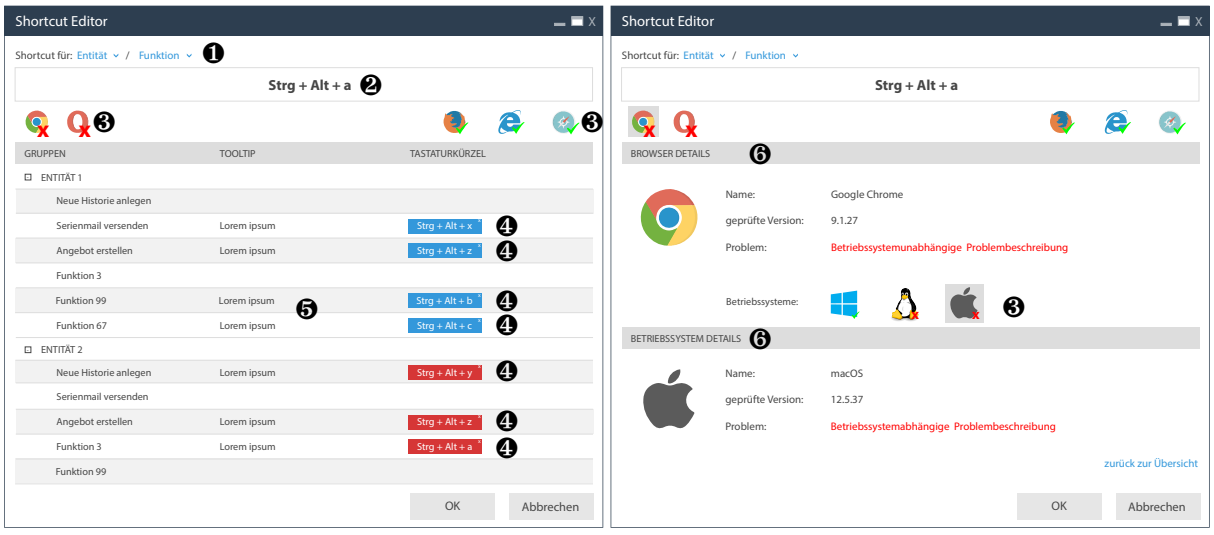


Abbildung 5: Designentwurf des UX-Designers

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 gespeichert 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.

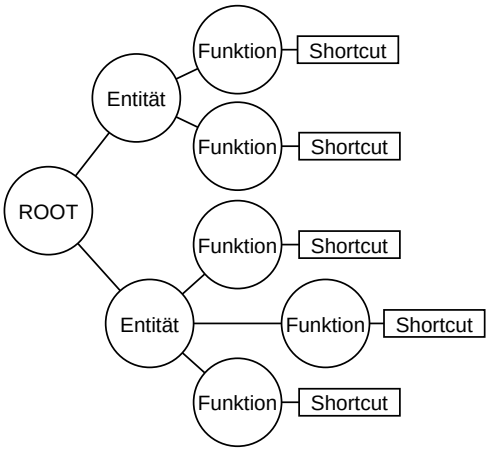


Abbildung 6: Baumstruktur für Entitäten und Funktionen

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 7).

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.

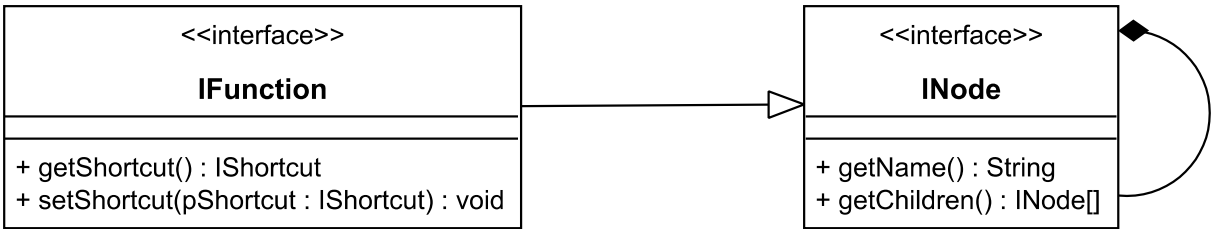


Abbildung 7: Klassendiagramm des Datenmodells für Entitäten und Funktionen

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 konfortabler 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

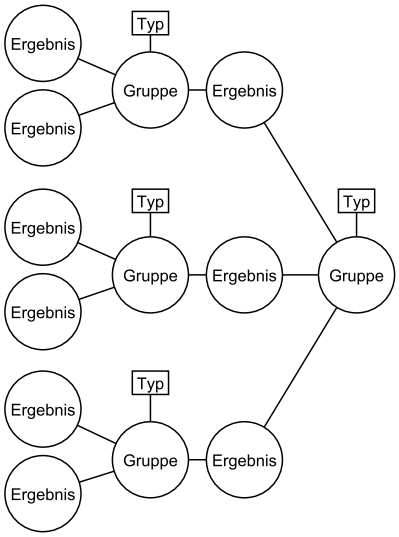


Abbildung 8: Baumstruktur für Browsertestergebnisse

Auch die Browserergebnisse werden als Baumstruktur gehalten. Allerdings 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 9 ist ein UML-Klassendiagramm eines Datenmodells abgebildet, welches den Anforderungen zum Halten von Browsertestergebnissen gerecht wird.

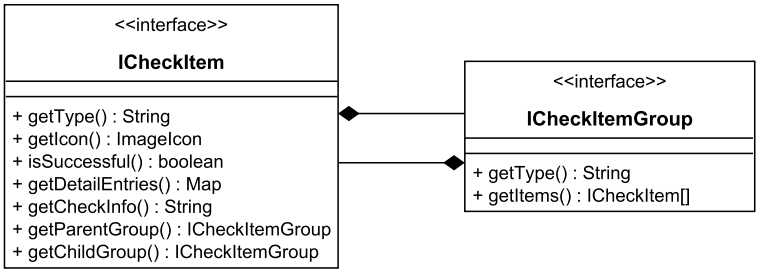


Abbildung 9: Klassendiagramm des Datenmodells für Browsertestergebnisse

In diesem Modell stellt ein **ICheckItem** ein einzelnes Testergbnis dar. Eine **ICheckItemGroup** hält neben einer beliebigen Anzahl von **ICheckItem**s einen Typ in Form eines **String**s. 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 zu speichern.

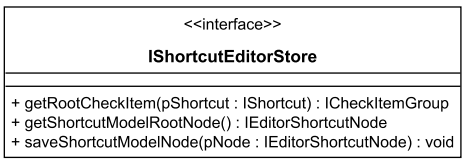


Abbildung 10: Klassendiagramm des Datenstores

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

Im Nachfolgenden wird anhand der

5.1 Allgemein

5.1.1 IRemoteLoggerCheckPoint

Kapselt einen bereits vorhandenen CheckPoint. Enthält allerdings noch die Informationen über den Grund (getCause()), warum dieser CheckPoint aufgetreten ist. Somit wird eine Baumstruktur aus IRemoteLoggerCheckPoints erzeugt, mit der sich auftretende Fehler leicht identifizieren und beheben lassen.

Definition innerhalb von ADITO4: Ein CheckPoint setzt sich aus seiner Art, seinem Modul, seiner Priorität, seinem Identifier, seinem Programm und einer für den Benutzer lesbaren Meldung zusammen.

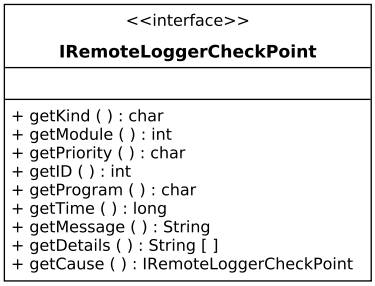


Abbildung 11: Aufbau eines „IRemoteLoggerCheckPoint“

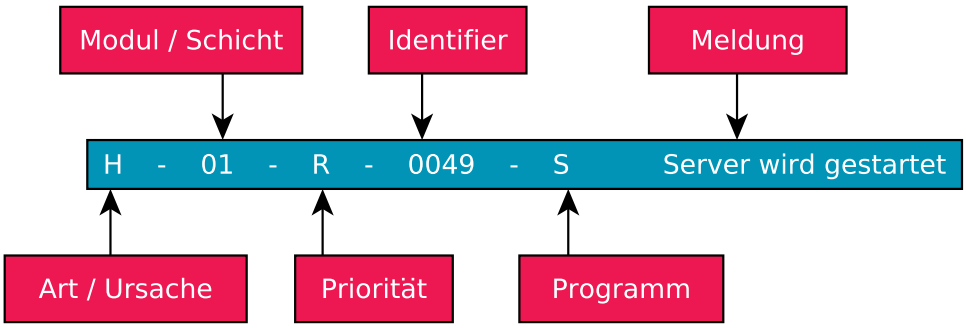


Abbildung 12: Aufbau eines CheckPoints in ADITO4

1. Art/Ursache	Gibt die Ursache des Fehlers an (Bug, Anwender- / Netzwerkfehler, etc.)
2. Modul/Schicht	Gibt an, in welcher Schicht der Fehler aufgetreten ist (DB-, Kommunkations-, Kalenderschicht, etc.)
3. Priorität	Gibt an, wie schwerwiegend die Meldung ist (A - Z, A = höchste Priorität)
4. Identifier	Jede Meldung hat eine 4-stellige ID, die innerhalb eines Modules eindeutig ist
5. Programm	Gibt an, welches Teilprogramm die Meldung verursachte (Client, Server, Designer, etc.)
6. Meldung	Eine für den Benutzer lesbare Beschreibung

Abbildung 13: Beschreibung der verschiedenen Bestandteile eines ADITO-CheckPoints

DefaultRemoteLoggerCheckPoint

Als Datencontainer steht der „DefaultRemoteLoggerCheckPoint“ zur Verfügung, um das o.g. Interface mit passenden Daten zu versorgen. Im Konstruktor erhält dieser ein Array aus CheckPoints und den zugehörigen Zeitstempel in Millisekunden. Die Arrayreihenfolge entspricht der zeitlichen Abfolge voneinander abhängigen Ereignissen („'Meldung 1' verursacht durch 'Meldung 2' verursacht durch 'Meldung 3' ... “). An der Stelle 0 enthält das Array den CheckPoint, dessen Daten durch die aktuelle „DefaultRemoteLoggerCheckPoint“-Instanz abgebildet werden sollen. An darauf folgender Stelle sind die Daten der übergeordneten CheckPoints, den „Causes“, zu finden.
Um den direkten Vorgänger instantiieren zu können benötigt man die Methode „initCause(...)“:

```
125 protected IRemoteLoggerCheckPoint initCause(CheckPoint[] pTrace, long pTime)
126 {
127   CheckPoint[] newTrace = Arrays.copyOfRange(pTrace, 1, pTrace.length);
128   return pTrace.length == 1 ? null : new DefaultRemoteLoggerCheckPoint(newTrace, pTime);
129 }
```

Abbildung 14: initCause(...)-Methodenimplementierung innerhalb des „DefaultRemoteLoggerCheckPoint“

TranslateableRemoteLoggerCheckPoint

In der Theorie kann jeder verbundene Remote-Logger-Client eine andere Sprache besitzen, unabhängig der des Remote-Logger-Servers. Da dies mit o.g. Konstrukt noch nicht möglich ist, kommt hier der „TranslateableRemoteLoggerCheckPoint“ ins Spiel. Dieser erweitert den „DefaultRemoteLoggerCheckPoint“ um eine Übersetzungsfunktion.

Er enthält die Methode „retranslate(pNewLocale : Locale)“ die aufgerufen werden kann, wenn die hinter dem CheckPoint liegende Meldung (siehe Abbildung 13, Punkt 6) in eine andere Sprache übersetzt werden soll. Falls diese in der gewünschten Sprache nicht vorliegt wird versucht, die englische Version zu laden. (Quellcode siehe Anhang **??**)

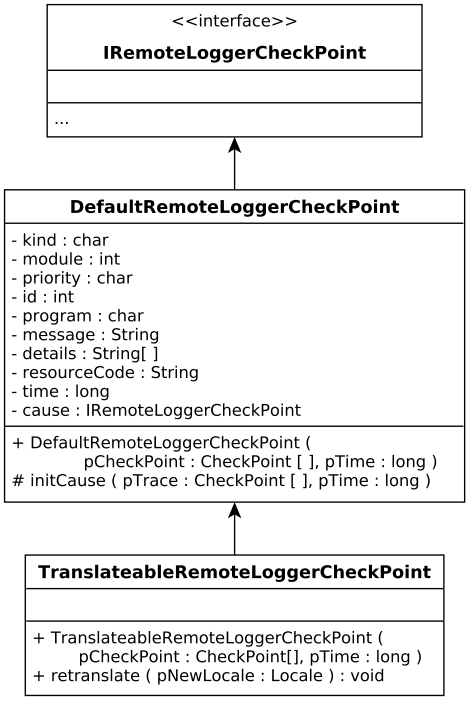


Abbildung 15: Klassenhierarchie des „IRemoteLoggerCheckPoint“

5.1.2 IRemoteLoggerCommand

Ein Remote-Logger-Kommando ist ein Befehl, der vom Remote-Logger-Client an den -Server gesendet wird um dort eine bestimmte Aktion auszuführen. Eine Instanz des IRemoteLoggerCommands muss vollständig serialisierbar sein und die kompletten Daten enthalten, die zur Durchführung der am Server hinterlegten Aktion benötigt werden (**??**).

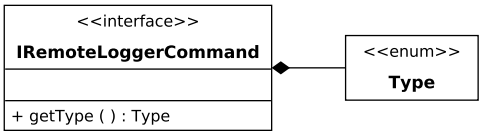


Abbildung 16: Klassendiagramm des „IRemoteLoggerCommand“

AuthorizationCommand

Mit Hilfe eines IRemoteLoggerCommands lässt sich ein Kommando zur Authorisierung eines Remote-Logger-Clients mittels beliebigen Login-Informationen (Username und Passwort, RSA-Schlüssel, etc.) am Remote-Logger-Server implementieren. Daraus ergeben sich mehrere Vorteile: Zum einen ist es durch diesen Authorisierungsvorgang gewährleistet, dass Unberechtigte keine Daten vom Remote-Logger-Server erhalten. Zum anderen können die schon vorhandenen Login-Informationen des ADITO4-Managers dazu verwendet werden, denn dieser muss sich vor dem Verbinden mit dem ADITO4-Server ebenso authentifizieren (siehe Abbildung **??**).

LanguageCommand

Ein weiteres Kommando stellt das „LanguageCommand“ dar. Dadurch ist es möglich, die Sprache, in der etwaige CheckPoint-Meldungen übersetzt werden, spezifisch für jeden Remote-Logger-Client separat festzulegen.

5.1.3 ObjectInputStreamConsumer

Der „ObjectInputStreamConsumer“ kapselt einen übergebenen InputStream in einen ObjectInputStream und liest so lange Daten in einem isolierten Thread aus, bis entweder ein Herunterfahren von außen angefragt wurde oder ein interner Fehler beim Auslesen aufgetreten ist. Im Konstruktor erwartet diese Klasse neben dem auszulesenden Stream, dem gewünschten Threadnamen, einen Consumer zur Verarbeitung erfolgreich ausgelesener Objekte. Durch das Java-Generic wird bestimmt, welchen Typ die auszulesenden Objekte vorweisen. Zusätzlich kann eine Funktion übergeben werden die anhand von intern aufgetretenen Fehlermeldungen entscheiden kann, ob der Auslesevorgang erneut gestartet oder der StreamConsumer geschlossen werden soll. Mit der „consume(...)“Methode kann ein neuer ObjectInputStreamConsumer erstellt werden.

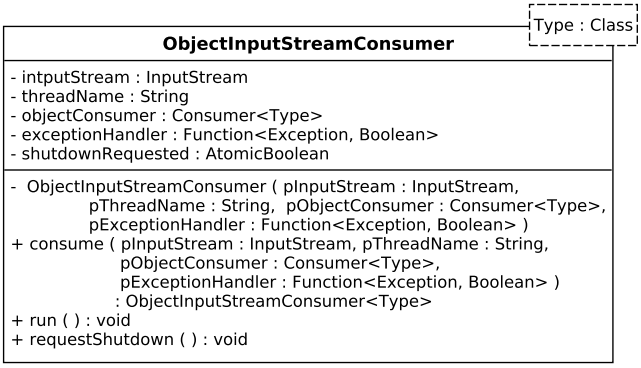


Abbildung 17: Aufbau des „ObjectInputStreamConsumer“

Der Hauptteil des o.g. Algorithmus ist in der „run()“Methode des Consumers implementiert:

```
56
57   while(!shutdownRequested.get())
58   {
59     ...
60     Object read;
61     while (!shutdownRequested.get() &&
62            (read = new ObjectInputStream(inputStream).readObject()) != null)
63       objectConsumer.accept((Type) read);
64     ...
65   }
```

Abbildung 18: Auslesen eines ObjectInputStreams im ObjectInputStreamConsumer

5.2 Serverseitig

5.2.1 RemoteLogger

Diese Klasse wird im bisher vorhandenen Logging-Modul von ADITO4 registriert und dient als Einstiegspunkt des Remote-Loggings. Über den Konstruktor wird die Priorität des Logger bestimmt. Diese besagt, ab welcher Dringlichkeitsstufe Meldungen geloggt werden dürfen. Ebenso werden hier Hostadresse und Hostport zugewiesen. Um später die eingehenden Login-Anfragen der Remote-Logger-Clients annehmen/ablehnen zu können, benötigt man zusätzlich eine „IRemoteLoggerLoginFacade“ (siehe 5.2.2), die ebenso im Konstruktor ihren Platz findet.

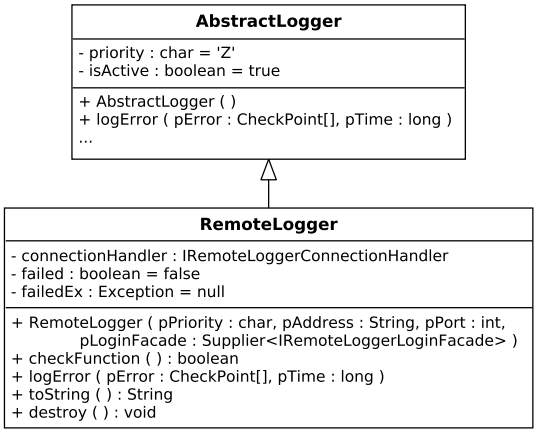


Abbildung 19: Aufbau des „RemoteLogger“

Der Remote-Logger erbt von der abstrakten Klasse „AbstractLogger“ wodurch es möglich ist, mit der Methode „logError(...)“ auf vom System übergebene CheckPoint-Arrays zu reagieren. Diese übergebenen Arrays sind jedoch noch nicht serialisierbar und können somit nicht über das Netzwerk geschickt werden. Eine Kapselung der Meldung in ein serialisierbares Objekt ist hier notwendig. Ebenso soll es möglich sein, die Meldung eines CheckPoints abhängig von der verbindungsspezifischen Sprache zu übersetzen. Hierzu dient das Interface „IRemoteLoggerCheckPoint“ und ihre Implementierung „TranslateableRemoteLoggerCheckPoint“ (siehe 5.1.1).

Sobald die Umwandlung der CheckPoints abgeschlossen ist, werden diese dem „connectionHandler“ übergeben, damit die Meldungen über das Netzwerk zu den derzeit angemeldeten Remote-Logger-Clients versendet werden können.

5.2.2 IRemoteLoggerLoginFacade

Dieses Interface kapselt einen Teil des ADITO4-Benutzer-Frameworks (siehe Abbildung **??**) für den Remote-Logger-Server. Dadurch lassen sich die vom Remote-Logger-Client erhaltenen Logininformationen (siehe Abbildung **??**, Schritt 3) auf Gültigkeit prüfen.

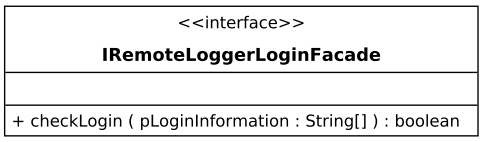


Abbildung 20: Aufbau der „LoginFacade“

Um einen Loginversuch zu validieren, übergibt man der im Interface definierten Methode „checkLogin(...)“ die zu prüfenden Informationen und man erhält als Rückgabewert, ob diese gültig sind.

```
25 @Override
26 public boolean checkLogin(@NotNull String[] pLoginInformation)
27 {
28   ManagerLoginUtil.LoginResult result = ManagerLoginUtil.checkLogin(pLoginInformation,
29                                         securityPrefs, loginPrefs, locale, userDirectory);
30
31   // Überprüft gleichzeitig, ob der hinter den Login-Informationen liegende Benutzer
32   // die passende Rolle besitzt.
33   return result.wasOK() && UserUtility.hasRole(result.getUser(), InternalRoles.ADMIN);
34 }
```

Abbildung 21: Implementierung der „checkLogin(...)“-Methode (siehe Anhang **??**)

5.2.3 IRemoteLoggerConnectionHandler

Der in Punkt 5.2.1 erwähnte „ConnectionHandler“ spezifiziert durch das Interface „IRemoteLoggerConnectionHandler“ und implementiert in der Klasse „RemoteLoggerServerConnectionHandler“ erhält mit der Methode „writeCheckPoint(...)“ die Anweisung, einen IRemoteLoggerCheckPoint (5.1.1) an alle derzeit verbundenen und autorisierten Remote-Logger-Clients zu senden. Ebenso besitzt er die Aufgabe, eingehende Verbindungsanfragen von neuen Clients zu verarbeiten und zu speichern.

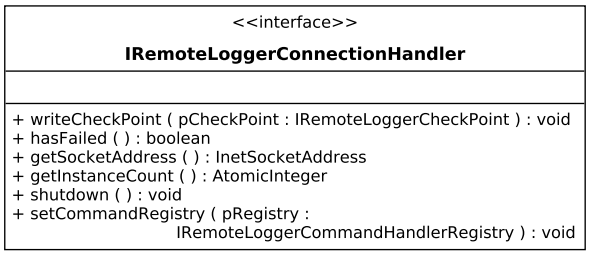


Abbildung 22: Aufbau des „IRemoteLoggerConnectionHandler“

Beide Aufgaben sind getrennt implementiert:
Auf der einen Seite wird jeder im ADITO4-System aufgetretene CheckPoint an alle autorisierten Verbindung innerhalb der o.g. Verbindungsliste gesendet.
Auf der anderen Seite läuft ein isolierter Hintergrundprozess (siehe Anhang **??**), der auf die Verbindung eines Remote-Logger-Clients wartet. Diese Verbindungsanfrage wird in Zusammenarbeit mit einem „Selector“ abgearbeitet und in eine „IRemoteLoggerServerConnection“ gekapselt. Das resultierende Verbindungsobjekt wird in der Verbindungsliste abgelegt.

Wenn man nun beide Aufgaben kombiniert betrachtet fällt auf, dass diese asynchron ablaufen müssen. Beispielsweise könnte sich ein Remote-Logger-Client verbinden, während der „ConnectionHandler“ CheckPoints an alle ihm bekannten Verbindungen nach außen sendet.
Hier kommt das Java-Schlüsselwort **synchronized** zum Einsatz. Dies bewirkt, dass keine konkurrierenden Zugriffe auf das gleiche Objekt gemacht werden können, sondern anfallende Zugriffe sequentiell abgearbeitet werden müssen. Dadurch kann sichergestellt werden, dass die o.g. Verbindungsliste während dem Senden der Meldungen nicht manipuliert wird.

5.2.4 IRemoteLoggerServerConnection

Die Verbindung zwischen einem Remote-Logger-Server und Remote-Logger-Client wird hier repräsentiert. Diese Klasse ermöglicht es unter anderem, einen IRemoteLogger-CheckPoint in der für den jeweiligen Client passenden Sprache zu senden. Dieses Senden

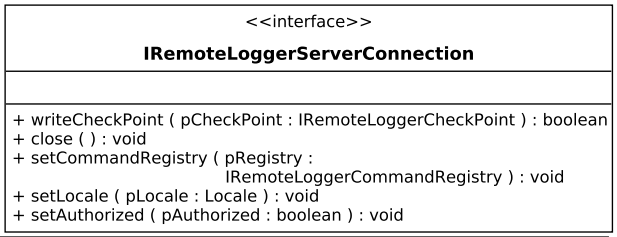


Abbildung 23: Aufbau des Interfaces „IRemoteLoggerServerConnection“

darf allerdings nur erfolgen, wenn die Verbindung davor durch ein IRemoteLogger-Command autorisiert wurde (siehe 5.1.2).

Ebenso werden hier etwaige empfangene Remote-Logger-Kommandos interpretiert und die zugehörigen „IRemoteLoggerCommandHandler“ ausgeführt. Diese Kommunikationsschnittstelle zwischen Remote-Logger-Client und Remote-Logger-Server ist durch den in Punkt 5.1.3 angesprochenen ObjectInputStreamConsumer realisiert:

```
23   ObjectInputStreamConsumer.consume(Channels.newInputStream(pChannel), null,
24                                     this::_handleCommand, this::_handleException);
```

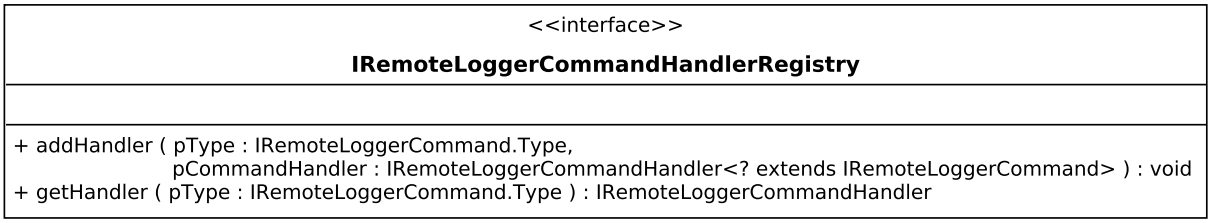
Abbildung 24: ObjectInputStreamConsumer filtert den InputStream nach IRemoteLoggerCommands

Das Senden von IRemoteLoggerCheckPoints innerhalb der Methode „writeCheckPoint(...)“ ist in der zugehörigen Implementierung durch einen ObjectOutputStream realisiert :

```
45 ByteArrayOutputStream baos = new ByteArrayOutputStream();
46 ObjectOutputStream oos = new ObjectOutputStream(baos);
47 oos.writeObject(pCheckpoint);
48 oos.flush();
49
50 channel.write(ByteBuffer.wrap(baos.toByteArray()));
```

Abbildung 25: Implementierung des Sendens von CheckPoints mittels einem ObjectOutputStream

5.2.5 IRemoteLoggerCommandHandlerRegistry



In dieser Klasse lassen sich IRemoteLoggerCommandHandler mit ihren zugehörigen IRemoteLoggerCommands verknüpfen. Falls ein spezifischer CommandHandler für das Abarbeiten von IRemoteLoggerCommands benötigt wird, kann man diesen mit der getHandler(...)-Methode ermitteln.

5.2.6 IRemoteLoggerCommandHandler

Verarbeitet ein IRemoteLoggerCommand (5.1.2), das vom Remote-Logger-Client zum Remote-Logger-Server gesendet wurde. Der Handler erhält hierzu über die „handle(...)“-Methode die IRemoteLoggerServerConnection, über die das Kommando empfangen wurde, und die Instanz des empfangenen Kommandos.

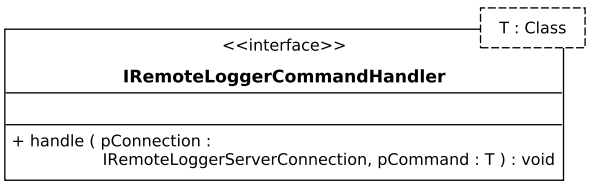


Abbildung 26: Aufbau des Interfaces „IRemoteLoggerCommandHandler“

Als Beispiel für einen CommandHandler ist der „AuthorizationCommandHandler“ (siehe Anhang **??**) zu nennen. Diesem wird im Konstruktor die in Punkt 5.2.2 erwähnte Login-Facade übergeben, um die empfangenen Login-Informationen auf Richtigkeit zu prüfen. Wenn der Login gültig ist, wird die Verbindungsinstanz als autorisiert gekennzeichnet, andernfalls wird die Verbindung sofort geschlossen.

```
28 @Override
29 public void handle(IRemoteLoggerServerConnection pConnection, AuthorizationCommand pCommand)
30 {
31   IRemoteLoggerLoginFacade loginFacade = null;
32   if(loginFacadeSupplier != null)
33     loginFacade = loginFacadeSupplier.get();
34
35   if (loginFacade == null || loginFacade.checkLogin(pCommand.getLoginInformation()))
36     pConnection.setAuthorized(true);
37   else
38     pConnection.close();
39 }
```

Abbildung 27: „handle(...)“-Methode des „AuthorizationCommandHandlers“

5.3 Client- / Managerseitig

5.3.1 IRemoteLoggerClientConnection

Kapselt eine Verbindung zum Remote-Logger-Server. Der Verbindungsaufbau erfolgt in der „connect()“-Methode mit Hilfe der Java-eigenen Netzwerk-API. Dadurch erhält man einen InputStream und der ObjectInputStreamConsumer (siehe 5.1.3) kann seine Arbeit beginnen.

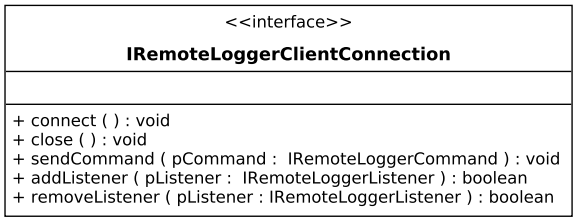


Abbildung 28: Aufbau des Interfaces „IRemoteLoggerClientConnection“

```
32 clientSocket = new Socket(host, port);
33 streamConsumer = ObjectInputStreamConsumer.consume(clientSocket.getInputStream(), null,
34                                 this::_fireCheckPointReceived, this::_handleException);
```

Abbildung 29: Initiieren der Verbindung zum Remote-Logger-Server am Remote-Logger-Client

Falls vom Server ein „IRemoteLoggerCheckPoint“-Objekt empfangen wird, so wird die private Methode „_fireCheckPointReceived(...)“ aufgerufen. Diese iteriert die Liste der derzeit registrierten IRemoteLoggerListener (siehe 5.3.2) und gibt den empfangenen IRemoteLoggerCheckPoint an jeden Listener weiter.

Ebenso ist es mit einer IRemoteLoggerClientConnection möglich, Kommandos an den Remote-Logger-Server zu senden (siehe 5.1.2).

5.3.2 IRemoteLoggerListener

Beschreibt einen Listener der zum einen aufgerufen wird, wenn sich der Status der Verbindung (Neue Verbindung, Verbindung getrennt) ändert und zum anderen wenn ein CheckPoint vom Remote-Logger-Server empfangen wurde.

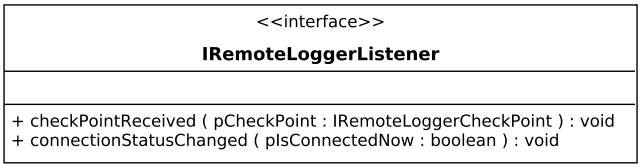


Abbildung 30: Aufbau des Interfaces „IRemoteLoggerListener“

5.3.3 IRemoteLoggerClientConnectionManager

Enthält und steuert eine IRemoteLoggerClient-Connection (siehe 5.3.1). Dieser Manager bildet die zentrale Stelle der Verbindung zwischen Remote-Logger-Client und Remote-Logger-Server und sorgt dafür, dass die Verbindung gekapselt bleibt. Die „connect(...)“-Methode übernimmt einen zusätzlichen Parameter: Die Sprache der Verbindung (siehe 5.1.2 - LanguageCommand).

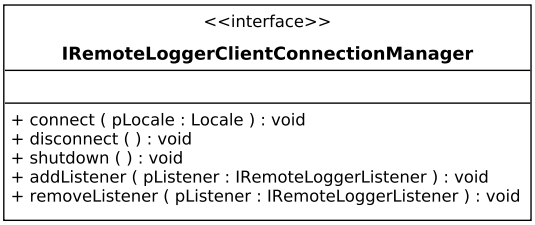


Abbildung 31: Aufbau des Interfaces „IRemoteLoggerListener“

Aufgrund des Sicherheitsaspekts ist das Senden von beliebigen Kommandos ab dieser Klasse von außen nicht mehr möglich. So sind Objekte an die Funktionalität des „Connection-Managers“ gebunden und können keine eigenen Methoden hinzufügen.
Ein zentrales Registrieren von IRemoteLoggerListener ist hier ebenfalls möglich (siehe 5.3.2). Das hat den Vorteil, dass beim Wechsel der IRemoteLoggerClientConnection-Instanz die Listener automatisch von der alten Verbindung entfernt und zur neuen Verbindung hinzugefügt werden.

5.3.4 Graphical User Interface

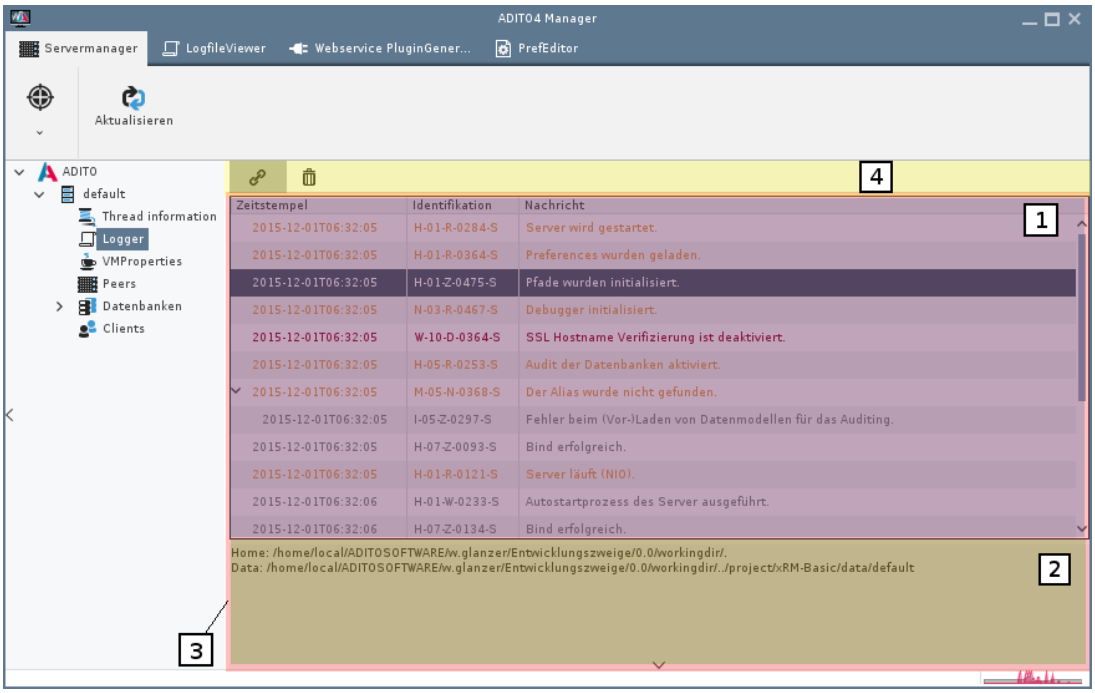


Abbildung 32: Remote-Logger-Client innerhalb des ADITO4-Managers

1. CheckPoint-Übersicht

Die TreeTable entspricht der Hauptkomponente des Frames. Hier werden alle empfangenen CheckPoints farblich aufbereitet und dem Benutzer als Kombination von Baum und Tabelle präsentiert.
In der ersten Spalte befindet sich der Zeitstempel der besagt, zu welchem Zeitpunkt der CheckPoint aufgetreten ist. Die darauf folgende Spalte enthält den Identifikator, damit der jeweilige CheckPoint eindeutig im kompletten System zu identifizieren ist. Die hinter dem CheckPoint liegende, für den Benutzer lesbare Nachricht ist in der letzten Spalte platziert. Diese wird bereits in der richtigen Sprache vom Remote-Logger-Server geliefert.

2. Detailansicht

In dieser Komponente (Instanz einer JTextArea) werden etwaige Details zu einem derzeit im Baum (1) selektierten CheckPoint dargestellt.

3. Container

Als Container für o.g. Komponenten dient eine Abwandlung der JSplitPane. Allgemein bietet eine SplitPane Platz für zwei Komponenten, wobei sich mittels einem Trenner der Platz interaktiv durch den Benutzer aufteilen lässt. Im ADITO4-Manager wird nicht die originale JSplitPane benutzt, sondern die ADITO-spezifische „OperaSplitpane“, da diese die JSplitPane um eine Aufklappfunktion erweitert. Somit wird es dem Benutzer ermöglicht, die komplette zweite Komponente durch einen Button am unteren Rand unsichtbar zu schalten.

4. Toolbar

Die Toolbar besitzt derzeit zwei Funktionen: Einerseits enthält sie eine Button, der bei Klick versucht die Verbindung zum Remote-Logger-Server aufzubauen, andererseits leert der zweite Button die Liste aller empfangenen CheckPoints, um somit eine verbesserte Übersichtlichkeit zu schaffen.

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 Kompiliervorgang 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 33: 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 5.2.2) 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 34: 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 Subklasse „_ConnectionManager“ (Abbildung 35) abgebildet ist. Diese enhält passenden Login-Informationen und die Verbindungsparameter, um sich erfolgreich mit dem vorher gestarteten

```
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 35: Verbindungsaufbau des Remote-Logger-Clients zum 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 36: Setzen eines empfangenen CheckPoints; Benachrichtigung der wartenden Threads

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

Abbildung 37: 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 36). 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 38: 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 39: 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. Beispielweise 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 5.1.2)
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

10 Literaturverzeichnis

- Riehle, Dirk (1996): „Entwurfsmuster“

11 Anhang

11.1 Anwendungsfälle

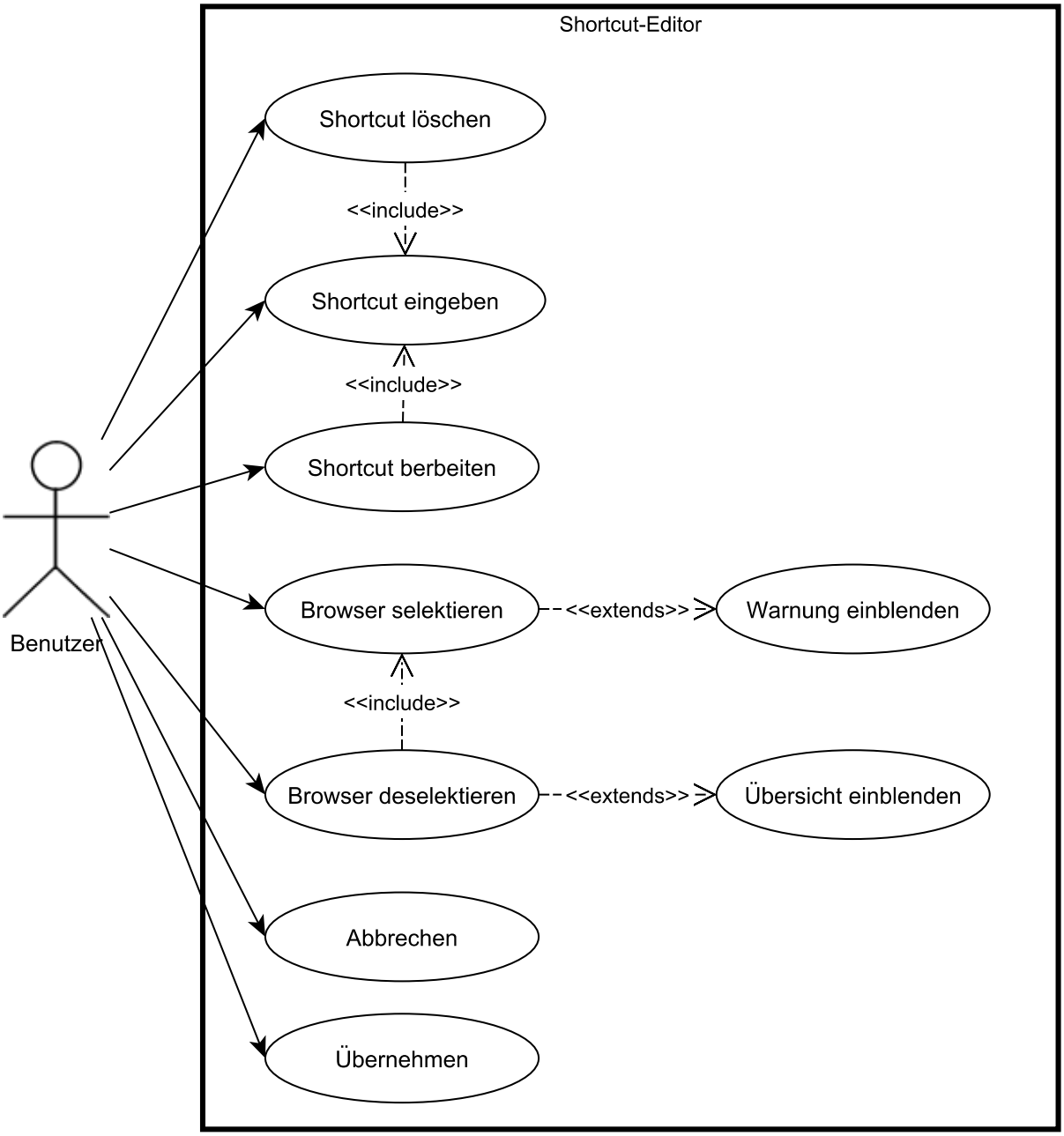


Abbildung 40: Use-Case-Diagramm

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 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