Abschlussprüfung Sommer 2018

Fachinformatiker für Anwendungsentwicklung

Dokumentation zur betrieblichen Projektarbeit

Shortcut-Editor

Implementierung eines Editors zur Bearbeitung von Tastaturkürzeln

Abgabetermin: 18.05.2018

Projektverantwortlicher: Robert Loipfinger

Prüfungsbewerber:
Korbinian Mifka
Pelzgartenstraße 12
84175 Johannesbrunn

Ausbildungsbetrieb:
ADITO Software GmbH
Konrad Zuse Str. 4
84144 Geisenhausen

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

Inhaltsverzeichnis

© Korbinian Mifka

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.

© Korbinian Mifka

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.

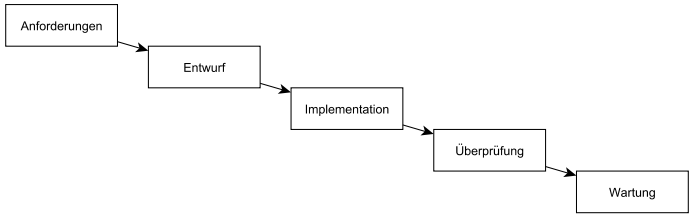


Abbildung 1: Wasserfallmodell

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	10
3. Implementierungsphase	45
4. Testphase	2
5. Dokumentationserstellung	10
	70

Abbildung 2: Grobe Zeitplanung

2.3 Ressourcenplanung

Im Zuge der Ressourcenplanung wurde eine Übersicht (siehe (XXX)) 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.

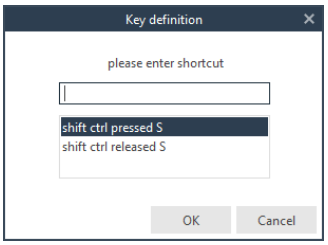


Abbildung 3: Bestehender Editor

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 (XXX) auf S.(X).

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)

© Korbinian Mifka

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.

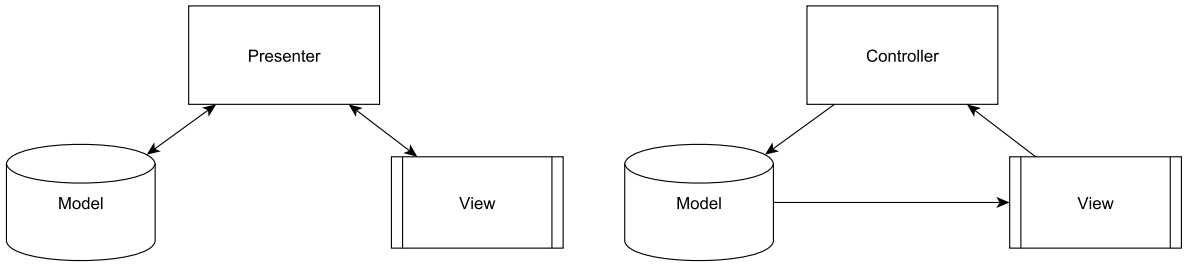


Abbildung 4: Model View Presenter vs. Model View Controller

Bei MVP lässt sich jede Komponente der Software einen 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 **??**). 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.

Im Folgenden wird näher auf die wichtigsten Bestandteile des Editors eingegangen und erläutert welchen Zweck diese haben:

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.

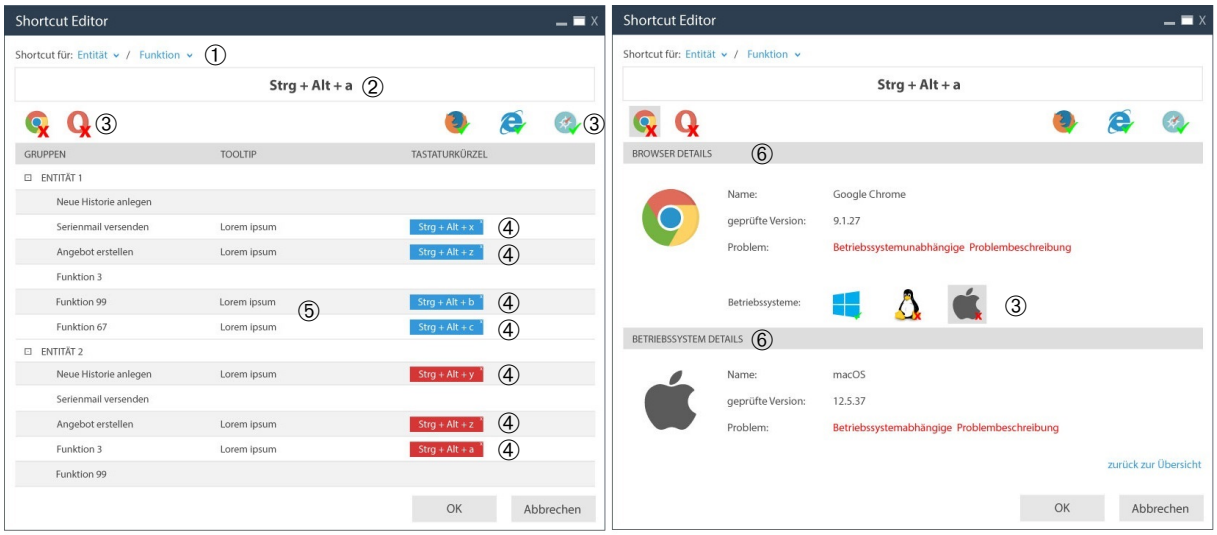
② **Shortcut-Field:** Hierbei handelt es sich um eine Komponente, welche die Darstellung und Bearbeitung von Shortcuts ermöglicht. Die Eingabe und Editierung des Tastaturkürzels kann nur bei selektiertem Zustand erfolgen. Demnach wechselt die Komponente bei Selektion in den Bearbeitungsmodus und verlässt 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 und wo dieser unbedenklich ist. 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 **Übersichtstabelle:** Die Tabelle dient wie die **Breadcrumb** auch der Navigation und bietet zudem einen Überblick über alle verfügbaren Shortcuts. Die Tastenkombinationen werden als Tag dargestellt.

⑥ **Accordion:** Ist ein **Check-Button** selektiert, so wird eine Accordion-Komponente angezeigt, welche detaillierte Informationen zu den Testergebnissen bietet. Um nur relevante Daten anzuzeigen, besteht die Möglichkeit einige Sektionen durch Klicken auf den Header einzuklappen.



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.

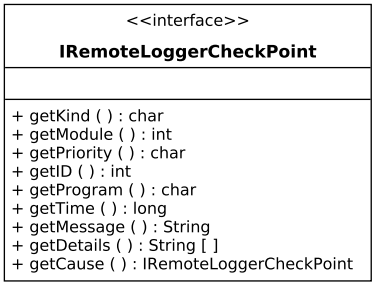


Abbildung 5: Aufbau eines „IRemoteLoggerCheckPoint“

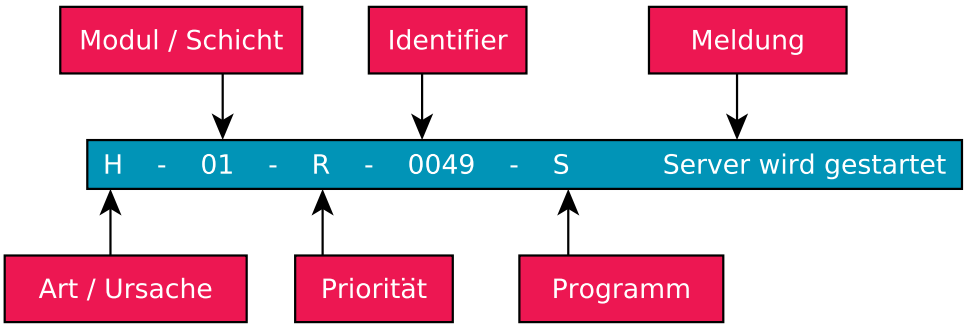


Abbildung 6: Aufbau eines CheckPoints in ADITO4

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 7: 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 8: 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 7, 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 11.1.1)

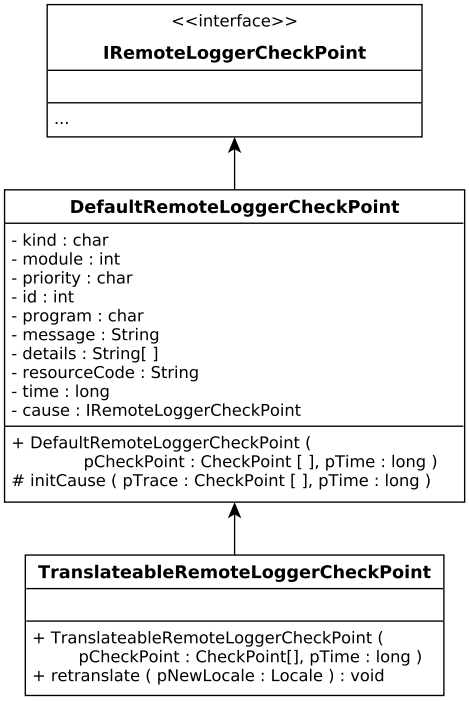


Abbildung 9: Klassenhierarchie des „IRemoteLoggerCheckPoint“

© Korbinian Mifka

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 (11.1.2).

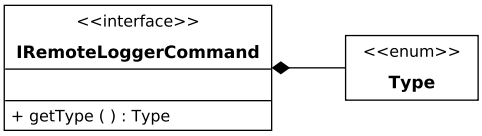


Abbildung 10: Klassendiagramm des „IRemoteLoggerCommand“

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.

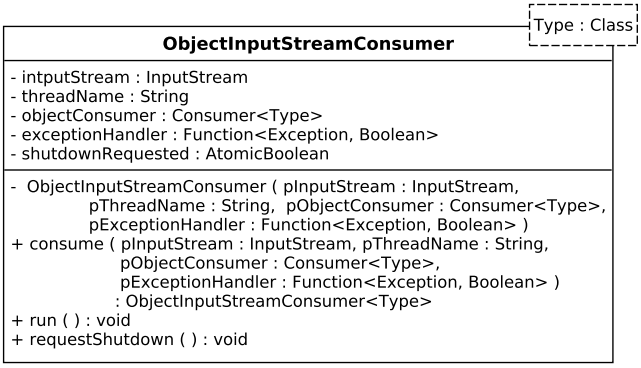


Abbildung 11: Aufbau des „ObjectInputStreamConsumer“

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 12: 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.

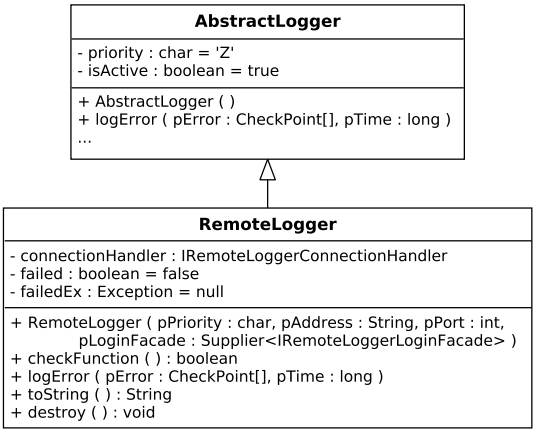


Abbildung 13: Aufbau des „RemoteLogger“

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.

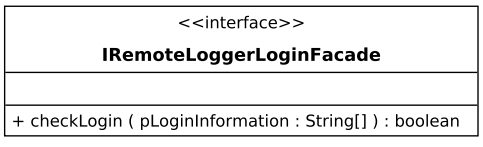


Abbildung 14: Aufbau der „LoginFacade“

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 15: Implementierung der „checkLogin(...)“-Methode (siehe Anhang 11.1.5)

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.

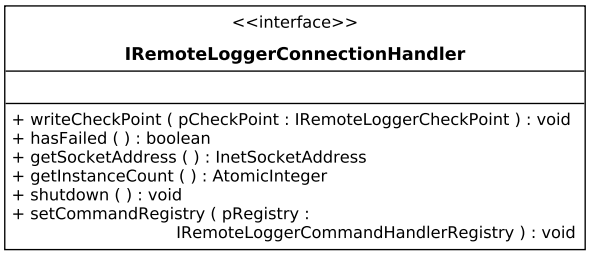


Abbildung 16: Aufbau des „IRemoteLoggerConnectionHandler“

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 11.1.6), 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

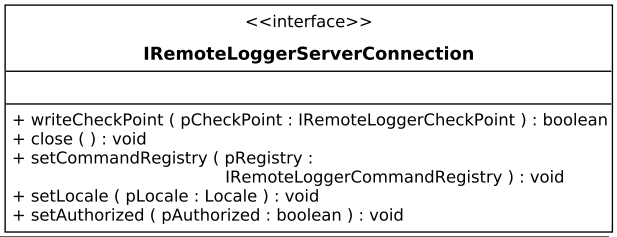


Abbildung 17: Aufbau des Interfaces „IRemoteLoggerServerConnection“

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);
```

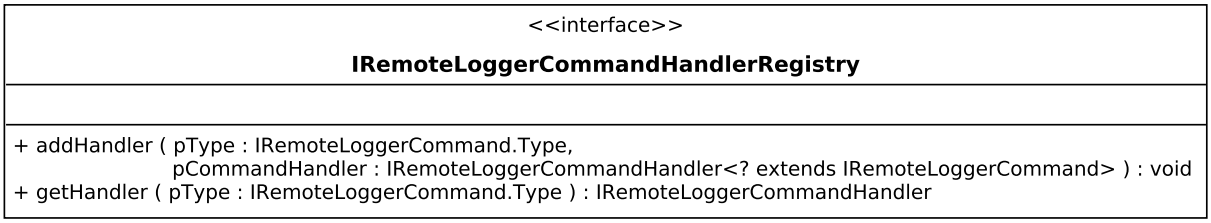
Abbildung 18: ObjectInputStreamConsumer filtert den InputStream nach IRemoteLoggerCommands

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 19: 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.

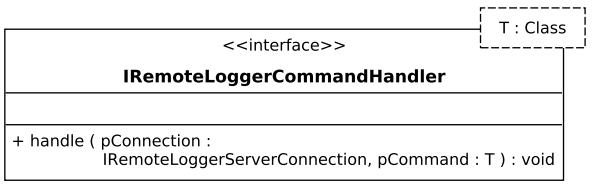


Abbildung 20: Aufbau des Interfaces „IRemoteLoggerCommandHandler“

© Korbinian Mifka

Als Beispiel für einen CommandHandler ist der „AuthorizationCommandHandler“ (siehe Anhang 11.1.2) 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 21: „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.

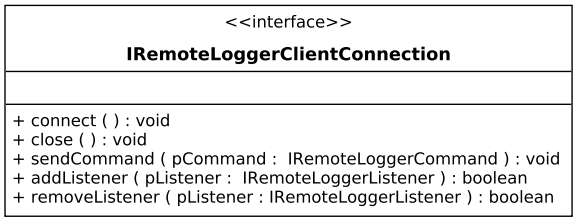


Abbildung 22: Aufbau des Interfaces „IRemoteLoggerClientConnection“

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

Abbildung 23: 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.

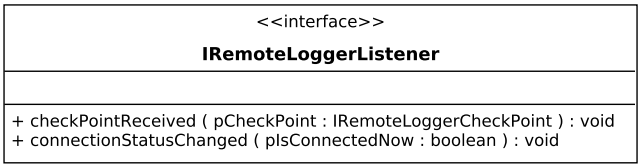


Abbildung 24: Aufbau des Interfaces „IRemoteLoggerListener“

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).

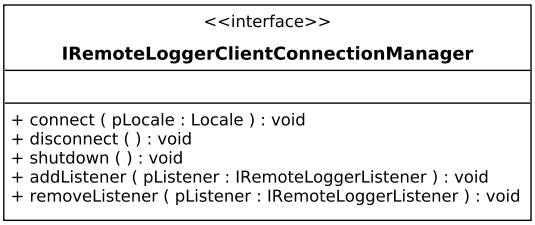


Abbildung 25: Aufbau des Interfaces „IRemoteLoggerListener“

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

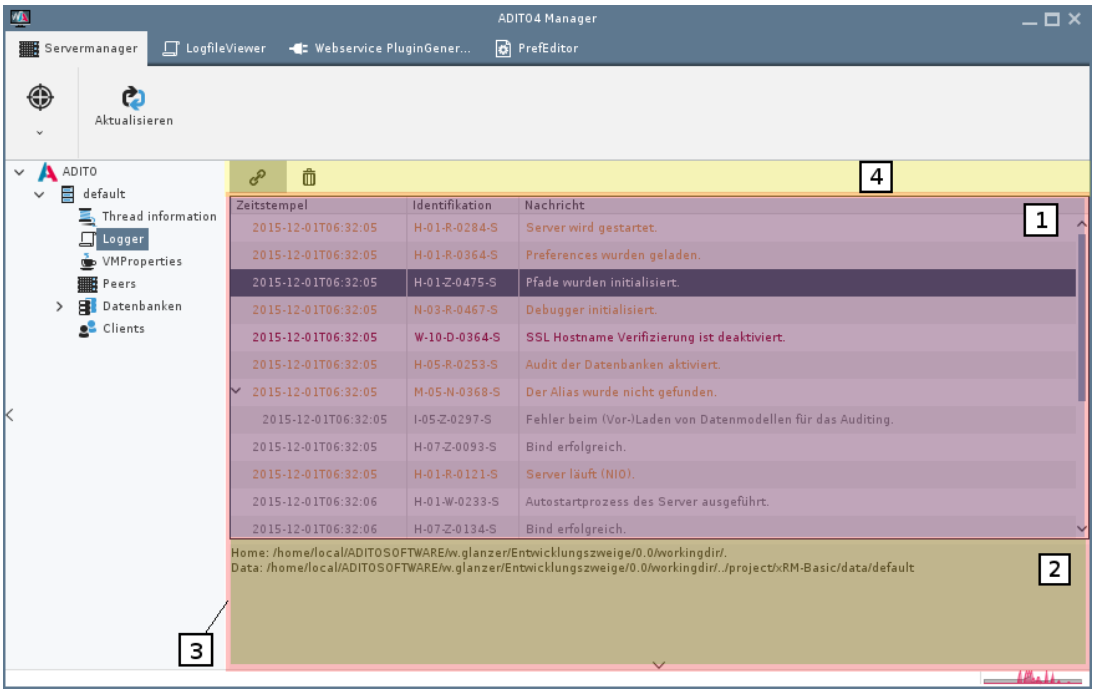


Abbildung 26: Remote-Logger-Client innerhalb des ADITO4-Managers

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 27: 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 28: 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 29) abgebildet ist. Diese enhält passenden Login-Informationen und die Verbindungsparameter, um sich erfolgreich mit dem vorher gestarteten Remote-Logger-Server

```
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 29: Verbindungsaufbau des Remote-Logger-Clients zum

© Korbinian Mifka

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 30: Setzen eines empfangenen CheckPoints; Benachrichtigung der wartenden Threads

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

Abbildung 31: 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 30). 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 32: 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 33: 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“

 © Korbinian Mifka

11 Anhang

11.1 Quellcode

11.1.1 TranslateableRemoteLoggerCheckPoint

```
/**
 * Erweitert den RemoteLoggerCheckPoint um eine
 * Funktion, diesen zu übersetzen
 *
 * @author W.Glanzer, 19.11.2015
 */
class TranslateableRemoteLoggerCheckPoint extends DefaultRemoteLoggerCheckPoint
  implements ITranslateableRemoteLoggerCheckPoint
{
  private String translatedMessage;

  /**
   * Wandelt einen CheckPoint in einen RemoteLoggerCheckPoint um
   *
   * @param pCheckPoint CheckPoint, der umgewandelt werden soll
   * @param pTime       Zeit, wann der übergebene CheckPoint aufgetreten ist
   */
  public TranslateableRemoteLoggerCheckPoint(CheckPoint[] pCheckPoint, long pTime)
  {
    super(pCheckPoint, pTime);
    translatedMessage = super.getMessage();
  }

  /**
   * Übersetzt den Meldungstext neu, in eine andere Sprache
   *
   * @param pNewLocale gewünschte Lokale, oder <tt>null</tt> für den Systemstandard
   */
  @Override
  public void retranslate(Locale pNewLocale)
  {
    String rc = getResourceCode();
    String[] details = getDetails();
    translatedMessage = Translator.translateCheckpoint(rc, details, pNewLocale);
  }

  @Override
  public String getMessage()
  {
    return translatedMessage;
  }

  @Override
  protected IRemoteLoggerCheckPoint initCause(CheckPoint[] pMyCheckpoint, long pTime)
  {
    CheckPoint[] newTrace = Arrays.copyOfRange(pMyCheckpoint, 1, pMyCheckpoint.length);
    if(pMyCheckpoint.length == 1)
      return null;

    return new TranslateableRemoteLoggerCheckPoint(newTrace, pTime);
  }
}
```

11.1.2 Implementierungen des Interfaces „IRemoteLoggerCommand“

```java
/**
* Hiermit lässt sich die Sprache einer
* Verbindung verändern, damit die
* CheckPoints in der vom Client
* gewählten Sprache ankommen
*
* @author W.Glanzer, 20.11.2015
*/
public class LanguageCommand implements
        IRemoteLoggerCommand, Serializable
{

  private Locale newLocale;

  public LanguageCommand(Locale pNewLocale)
  {
    newLocale = pNewLocale;
  }

  /**
  * Enthält die neue Sprache,
  * die der Client möchte
  *
  * @return Die gewünschte Sprache
  */
  public Locale getNewLocale()
  {
    return newLocale;
  }

  @NotNull
  @Override
  public Type getType()
  {
    return Type.LANGUAGE;
  }
}
```

```java
/**
 * Kommando zur Authorisierung eines
 * Remote-Logger-Clients auf dem
 * Remote-Logger-Server mit bestimmten
 * Login-Informationen
 *
 * @author W.Glanzer, 23.11.2015
 */
public class AuthorizationCommand
    implements IRemoteLoggerCommand
{

  private String[] loginInformation;

  public AuthorizationCommand(
        String[] pLoginInformation)
  {
    loginInformation = pLoginInformation;
  }

 /**
  * Enthält die LoginInformationen
  *
  * @return Array aus LoginInformationen
  */
  public String[] getLoginInformation()
  {
    return loginInformation;
  }

  @NotNull
  @Override
  public Type getType()
  {
    return Type.AUTHORIZE;
  }
}
```

11.1.3 ObjectInputStreamConsumer

```java
/**
 * Kapselt den übergebenen InputStream in einen ObjectInputStream.
 * Dieser liest so lange Daten in einem extra Thread aus,
 * bis ein shutdown angefragt wurde, oder ein Fehler aufgetreten ist
 * und der exceptionHandler einen Retry verweigert hat
 *
 * @author W.Glanzer, 20.11.2015
 */
public class ObjectInputStreamConsumer<Type> implements Runnable
{

  private final InputStream inputStream;
  private final String threadName;
  private final Consumer<Type> objectConsumer;
  private final Function<Exception, Boolean> exceptionHandler;
  private AtomicBoolean shutdownRequested = new AtomicBoolean();

  private ObjectInputStreamConsumer(@NotNull InputStream pInputStream,
    @Nullable String pThreadName, @NotNull Consumer<Type> pObjectConsumer,
    @Nullable Function<Exception, Boolean> pExceptionHandler)
  {
    inputStream = pInputStream;
    threadName = pThreadName;
    objectConsumer = pObjectConsumer;
    exceptionHandler = pExceptionHandler;
  }
```


© Korbinian Mifka

```
27    /**
28     * Erstellt einen ObjectInputStreamConsumer.
29     * Dieser liest so lange Daten aus, bis ein shutdown angefragt wurde, oder ein
30     * Fehler aufgetreten und der exceptionHandler einen Retry verweigert hat
31     *
32     * @param pIStream           InputStream, der ausgelesen werden soll.
33     * @param pTName             Name des Threads, in dem der ObjectInputStreamConsumer läuft
34     * @param pObjectConsumer    Erhält alle empfangenen Objekte des Consumers
35     * @param pExceptionHandler  Erhält alle aufgetretenen Exceptions.
36     *                           Wird in dieser Funktion <tt>true</tt> zurückgegeben, dann
37     *                           wird versucht, erneut vom Stream zu lesen.
38     *                           Bei <tt>false</tt> wird versucht, die Verbindung zu trennen.
39     * @return Den zugehörigen ObjectInputStreamConsumer
40     */
41    public static <T> ObjectInputStreamConsumer<T> consume(@NotNull InputStream pIStream,
42      @Nullable String pTName, @NotNull Consumer<T> pObjectConsumer,
43      @Nullable Function<Exception, Boolean> pExceptionHandler)
44    {
45      ObjectInputStreamConsumer<T> consumer =
46        new ObjectInputStreamConsumer<>(pIStream, pTName, pObjectConsumer, pExceptionHandler);
47      ThreadPool.getInstance().execute(consumer);
48      return consumer;
49    }
50    @Override
51    public void run()
52    {
53      if(threadName != null)
54        Thread.currentThread().setName(threadName);
55
56      while(!shutdownRequested.get())
57      {
58        Object read;
59        try
60        {
61          while ((read = new ObjectInputStream(inputStream).readObject()) != null)
62            && !shutdownRequested.get())
63          {
64            objectConsumer.accept((Type) read);
65          }
66        }
67        catch (Exception e)
68        {
69          if(exceptionHandler != null && !shutdownRequested.get())
70          {
71            if(!exceptionHandler.apply(e)) //Kein Retry gewünscht
72            {
73              try
74              {
75                requestShutdown();
76              }
77              catch (Exception ex)
78              {
79                exceptionHandler.apply(ex);
80              }
81            }
82
83            break;
84          }
85        }
86      }
87    }
88
89    /**
90     * Frägt an, dass sich der StreamConsumer bitte beenden würde.
91     * Der anfängliche InputStream wird geschlossen!
92     *
93     * @throws AditoIOException IOException wenn ein Fehler aufgetreten ist
94     */
95    public void requestShutdown() throws AditoIOException
96    {
97      try
98      {
99        shutdownRequested.set(true);
100       inputStream.close();
101     }
102     catch(Exception e)
103     {
104       // Stream konnte nicht geschlossen werden
105       throw new AditoIOException(e, 10, 403);
106     }
107   }
108 }
```

11.1.4 Remote-Logger

```java
/**
 * Logger, der Checkpoints zu verbunden RemoteLogger-Instanzen senden kann
 *
 * @author W.Glanzer, 18.11.2015
 */
public class RemoteLogger extends AbstractLogger
{
  private IRemoteLoggerConnectionHandler connectionHandler;

  private boolean failed = false;
  private Exception failedEx = null;

  /**
   * Erzeugt einen neuen RemoteLogger
   *
   * @param pPriority                die Priorität des Loggers
   * @param pAddress                 die Addresse des Listeners
   * @param pPort                    der Port des Listeners
   * @param pRemoteLoggerLoginFacade Facade zum einloggen von RemoteLogger-Clients
   *                                 am RemoteLogger-Server, oder <tt>null</tt> wenn
   *                                 generell alle Clients erlaubt sind
   */
  public RemoteLogger(char pPriority, String pAddress, int pPort,
    @Nullable Supplier<IRemoteLoggerLoginFacade> pRemoteLoggerLoginFacade)
  {
    try
    {
      IRemoteLoggerCommandHandlerRegistry commandRegistry =
        new MapRemoteLoggerCommandHandlerRegistry();
      commandRegistry.addHandler(IRemoteLoggerCommand.Type.AUTHORIZE,
        new AuthorizationCommandHandler(pRemoteLoggerLoginFacade));
      commandRegistry.addHandler(IRemoteLoggerCommand.Type.LANGUAGE,
        new LanguageCommandHandler());

      RemoteLoggerInstanceManager im = RemoteLoggerInstanceManager.getInstance();
      connectionHandler = im.getConnectionHandler(pAddress, pPort);
      connectionHandler.setCommandRegistry(commandRegistry);
      setPriority(pPriority);
    }
    catch (Exception ex)
    {
      System.err.println("RemoteLogger failed:");
      ex.printStackTrace();

      failed = true;
      failedEx = ex;
    }
  }

  /**
   * Hier kann der Logger einen Selbsttest machen. Schlägt dieser fehl
   * kann er eine neuen Fehler erzeugen damit andere Logger dies melden können.
   *
   * @return <tt>true</tt> wenn der Logger funktioniert, andernfalls <tt>false</tt>
   */
  public boolean checkFunction()
  {
    // Neuen Wert berechnen
    failed = failed || connectionHandler.hasFailed();

    if (failed)
    {
      AditoException ex = new AditoException(failedEx, 0, 32);
      CPH.checkPoint(ex);
    }

    return !failed;
  }

  /**
   * Loggt einen Checkpoint
   *
   * @param pError der Checkpoint
   * @param pTime  die Zeit, wann er aufgetreten ist
   */
  public void logError(CheckPoint[] pError, long pTime)
  {
```


© Korbinian Mifka

```
78      if (earlyBreak(pError))
79        return;
80      if (failed)
81        return;
82
83      IRemoteLoggerCheckPoint cp = new TranslateableRemoteLoggerCheckPoint(pError, pTime);
84      connectionHandler.writeCheckPoint(cp);
85    }
86
87    /**
88     * Zerstört diesen Logger
89     *
90     * @throws AditoException Wird geworfen, wenn dieser Logger
91     *                        nicht ordnungsgemäß beendet werden konnte
92     */
93    public synchronized void destroy() throws AditoException
94    {
95      super.destroy();
96      failed = true;
97      RemoteLoggerInstanceManager.getInstance().shutdownRemoteLoggerHandler(connectionHandler);
98    }
99
100   /**
101    * Erstellt eine String-Repräsentanz des RemoteLoggers
102    *
103    * @return Lesbarer String
104    */
105   public String toString()
106   {
107     return "RemoteLogger [" + priority + "]: " +
108       (connectionHandler == null ? "null" : connectionHandler.getSocketAddress());
109   }
110 }
```

11.1.5 Implementierung des Interfaces „IRemoteLoggerLoginFacade“

```
1  /**
2   * Login-Facade, damit sich der RemoteLogger-Client nur dann wirklich
3   * einloggen kann, wenn die Login-Daten stimmen!
4   *
5   * @author W.Glanzer, 26.11.2015
6   */
7  public class RemoteLoggerLoginFacadeImpl implements IRemoteLoggerLoginFacade
8  {
9    private final SecurityPrefs securityPrefs;
10   private final LoginPrefs loginPrefs;
11   private final Locale locale;
12   private final IUserDirectoryExt userDirectory;
13
14   public RemoteLoggerLoginFacadeImpl(@NotNull SecurityPrefs pSecurityPrefs,
15                                      @NotNull LoginPrefs pLoginPrefs,
16                                      @Nullable Locale pLocale,
17                                      IUserDirectoryExt pUserDirectory)
18   {
19     securityPrefs = pSecurityPrefs;
20     loginPrefs = pLoginPrefs;
21     locale = pLocale;
22     userDirectory = pUserDirectory;
23   }
24
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   }
35 }
```

11.1.6 Implementierung des Interfaces „IRemoteLoggerConnectionHandler“

```
/**
 * Sobald sich ein neuer RemoteLogger verbinden will springt dieser
 * Listener an, und erstellt eine neue DefaultRemoteLoggerServerConnection.
 * Diese speichert er im instances-Set ab, für spätere Benutzung
 */
private class _RemoteLoggerConnectionListener implements Runnable
{

  /**
   * Wird aufgerufen, wenn sich ein neuer Logger anmeldet
   */
  public void run()
  {
    Thread.currentThread().setName(IAditoThreads.REMOTELOGGERCONNECTIONLISTENER);
    while (!(failed || shutdown))
    {
      try
      {
        // Warten, bis sich ein Channel-Client verbinden will
        selector.select(); // Blockiert hier
        Iterator selectedKeys = selector.selectedKeys().iterator();
        while (selectedKeys.hasNext())
        {
          SelectionKey key = (SelectionKey) selectedKeys.next();
          selectedKeys.remove();

          if (!key.isValid())
            continue;

          if (key.isAcceptable())
          {
            ServerSocket serverSocket = ((ServerSocketChannel) key.channel()).socket();
            Socket clientSocket = serverSocket.accept();
            SocketDefaults.setSocketDefaults(clientSocket);
            SocketChannel chnl = clientSocket.getChannel();
            _connectionCreated(new DefaultRemoteLoggerServerConnection(chnl));
          }
        }
      }
      catch (Exception ex)
      {
        if (!shutdown)
          failed = true;
      }
    }
  }
  /**
   * Wird aufgerufen, wenn eine neue Connection aufgebaut wurde
   *
   * @param pNewConnection  Connection, die neu aufgebaut wurde
   */
  private void _connectionCreated(@NotNull IRemoteLoggerServerConnection pNewConnection)
  {
    pNewConnection.setCommandRegistry(commandRegistry);
    for (IRemoteLoggerCheckPoint currPoint : points)
    {
      pNewConnection.writeCheckPoint(currPoint);
    }
    synchronized (instances)
    {
      instances.add(pNewConnection);
    }
  }
}
```


© Korbinian Mifka

11.1.7 Implementierung des Interfaces „IRemoteLoggerServerConnection“

```
/**
 * Eine Instanz des Loggers, gekoppelt mit einem ObjectOutputStream
 * Jede Verbindung hat eine eigene Instanz
 *
 * @author W.Glanzer, 18.11.2015
 */
class DefaultRemoteLoggerServerConnection implements IRemoteLoggerServerConnection
{
  private SocketChannel channel;
  private Locale clientLocale;
  private IRemoteLoggerCommandHandlerRegistry registry;
  private boolean authorized;

  /**
   * Erzeugt eine neue Instanz
   *
   * @param pChannel der Netzwerkchannel der Instanz
   */
  public DefaultRemoteLoggerServerConnection(SocketChannel pChannel)
  {
    channel = pChannel;
    clientLocale = Locale.getDefault(); //Anfangswert
    ObjectInputStreamConsumer.consume(Channels.newInputStream(pChannel), null,
      this::_handleCommand, this::_handleException);
  }

  /**
   * Loggt einen für den RemoteLogger gekapselten Checkpoint
   *
   * @param pCheckpoint Checkpoint, der aufgetreten ist
   * @return <tt>true</tt> wenn erfolgreich
   */
  @Override
  public boolean writeCheckPoint(IRemoteLoggerCheckPoint pCheckpoint)
  {
    try
    {
      if (channel.isConnected() && pCheckpoint != null)
      {
        // Wenn nötig ab hier übersetzen, da nur die Connection weiß,
        // welche Sprache die verbundene GUI hat
        if(pCheckpoint instanceof ITranslateableRemoteLoggerCheckPoint)
          ((ITranslateableRemoteLoggerCheckPoint) pCheckpoint).retranslate(clientLocale);

        ByteArrayOutputStream baos = new ByteArrayOutputStream();
        ObjectOutputStream oos = new ObjectOutputStream(baos);
        oos.writeObject(pCheckpoint);
        oos.flush();

        channel.write(ByteBuffer.wrap(baos.toByteArray()));

        return true;
      }
      else
        return false;
    }
    catch (Exception ex)
    {
      // Wird geworfen, wenn die Verbindung wieder getrennt ist
      close();
      return false;
    }
  }

  /**
   * Setzt die CommandRegistry zur Verarbeitung von Kommandos
   *
   * @param pRegistry  Registry, die gesetzt werden soll, oder
   *                   <tt>null</tt> wenn alle Commands ignoriert werden sollen
   */
  @Override
  public void setCommandRegistry(@Nullable IRemoteLoggerCommandHandlerRegistry pRegistry)
  {
    registry = pRegistry;
  }

```

```
77
78    /**
79     * Setzt die Locale dieser Verbindung.
80     * Jede Connection kann ihre eigene Locale besitzen!
81     *
82     * @param pLocale  Locale, die gesetzt werden soll und
83     *                 die Sprache des Clients angibt
84     */
85    @Override
86    public void setLocale(@NotNull Locale pLocale)
87    {
88      clientLocale = pLocale;
89    }
90
91    @Override
92    public void setAuthorized(boolean pAuthorized)
93    {
94      authorized = pAuthorized;
95    }
96
97    /**
98     * Beendet die Instanz
99     */
100   @Override
101   public void close()
102   {
103     try
104     {
105       if (channel.isConnected())
106         channel.close();
107     }
108     catch (Exception ex)
109     {
110       System.err.println("DefaultRemoteLoggerServerConnection error[1]:");
111       ex.printStackTrace();
112     }
113   }
114 }
```

11.1.8 Implementierung des Interfaces „IRemoteLoggerCommandHandlerRegistry“

```
1  /**
2   * Implementierung einer CommandHandlerRegistry, bei der sich
3   * verschiedene CommandHandler einhängen lassen.
4   * Die Handler werden in einer Map gespeichert, und beim Aufruf
5   * von "shutdown" wieder gelöscht!
6   *
7   * @author W.Glanzer, 20.11.2015
8   * @see IRemoteLoggerCommandHandlerRegistry
9   */
10 class MapRemoteLoggerCommandHandlerRegistry implements IRemoteLoggerCommandHandlerRegistry
11 {
12
13   private final Map<IRemoteLoggerCommand.Type,
14    IRemoteLoggerCommandHandler<? extends IRemoteLoggerCommand>> handlers = new HashMap<>();
15
16   @Override
17   public void addHandler(IRemoteLoggerCommand.Type pType,
18    IRemoteLoggerCommandHandler<? extends IRemoteLoggerCommand> pCommandHandler)
19   {
20     synchronized (handlers)
21     {
22       handlers.put(pType, pCommandHandler);
23     }
24   }
25
26   @Nullable
27   @Override
28   public IRemoteLoggerCommandHandler getHandler(IRemoteLoggerCommand.Type pCommand)
29   {
30     synchronized (handlers)
31     {
32       return handlers.get(pCommand);
33     }
34   }
35 }
```


© Korbinian Mifka

11.1.9 Implementierungen des Interfaces „IRemoteLoggerCommandHandler“

```java
/**
 * Handlet das Kommando zur Autorisierung des RemoteLogger-Clients am RemoteLogger-Server.
 * Wenn die Autorisierung erfolgreich ist, dann wird dies der Verbindung gesetzt.
 * Andernfalls wird die Verbindung SOFORT geschlossen!
 *
 * @author W.Glanzer, 23.11.2015
 * @see AuthorizationCommand
 */
public class AuthorizationCommandHandler implements
  IRemoteLoggerCommandHandler<AuthorizationCommand>
{

  private Supplier<IRemoteLoggerLoginFacade> loginFacadeSupplier;

  /**
   * Erstellt einen CommandHandler zur Autorisierung vom RemoteLogger-Clients am Server
   *
   * @param pRemoteLoggerLoginFacade Facade zum einloggen von RemoteLogger-Clients am
   *                                 RemoteLogger-Server, oder <tt>null</tt> wenn
   *                                 generell alle Clients erlaubt sind
   */
  public AuthorizationCommandHandler(
    @Nullable Supplier<IRemoteLoggerLoginFacade> pRemoteLoggerLoginFacade)
  {
    loginFacadeSupplier = pRemoteLoggerLoginFacade;
  }

  @Override
  public void handle(IRemoteLoggerServerConnection pConnection, AuthorizationCommand pCmd)
  {
    IRemoteLoggerLoginFacade loginFacade = null;
    if(loginFacadeSupplier != null)
      loginFacade = loginFacadeSupplier.get()

    boolean loginOK = loginFacade == null ||
                      loginFacade.checkLogin(pCmd.getLoginInformation());
    if (loginOK)
      pConnection.setAuthorized(true);
    else
      pConnection.close();
  }

}
```

```java
/**
 * Handelt das "Language"-Kommando am Server ab
 * und setzt die neue Sprache der ServerConnection.
 * Somit muss dann nicht der Client die CheckPoints übersetzen,
 * sondern der Server schickt diese schon in der richtigen Sprache.
 *
 * @author W.Glanzer, 20.11.2015
 * @see LanguageCommand
 */
public class LanguageCommandHandler
  implements IRemoteLoggerCommandHandler<LanguageCommand>
{

  @Override
  public void handle(IRemoteLoggerServerConnection pConnection, LanguageCommand pCommand)
  {
    Locale newLocale = pCommand.getNewLocale();
    if(newLocale != null)
      pConnection.setLocale(newLocale);
  }

}
```

11.1.10 Implementierung des Interfaces „IRemoteLoggerClientConnection“

```java
/**
 * Verbindung RemoteLogger
 * Manager->Server
 *
 * @author W.Glanzer, 19.11.2015
 */
public class RemoteLoggerClientConnection implements IRemoteLoggerClientConnection
{
  private final List<IRemoteLoggerListener> loggerListenerList = new ArrayList<>();
  private final String host;
  private final int port;
  private String[] loginInformation;
  private boolean closed = false;
  private Socket clientSocket;
  private ObjectInputStreamConsumer<IRemoteLoggerCheckPoint> streamConsumer;

  public RemoteLoggerClientConnection(String pHost, int pPort, String[] pLoginInformation)
  {
    host = pHost;
    port = pPort;
    loginInformation = pLoginInformation;
  }

  @Override
  public void connect() throws AditoIOException
  {
    if ((clientSocket != null && !clientSocket.isClosed()) || closed)
      return;

    try
    {
      clientSocket = new Socket(host, port);
      streamConsumer = ObjectInputStreamConsumer.consume(clientSocket.getInputStream(),
        null, this::_fireCheckPointReceived, this::_handleException);

      // Einloggen
      sendCommand(new AuthorizationCommand(loginInformation));

      _fireConnectionStatusChanged(true);
    }
    catch (Exception e)
    {
      throw new AditoIOException(e, 16, 206);
    }
  }

  @Override
  public void close() throws AditoIOException
  {
    try
    {
      if (!closed)
      {
        streamConsumer.requestShutdown(); //schließt den inputStream schon
        closed = true;
        clientSocket = null;
        _fireConnectionStatusChanged(false);
      }
    }
    catch (Exception e)
    {
      throw new AditoIOException(e, 16, 207);
    }
  }

  @Override
  public void sendCommand(@NotNull IRemoteLoggerCommand pCommand) throws AditoIOException
  {
    try
    {
      if (!closed)
      {
        ByteArrayOutputStream outputstream = new ByteArrayOutputStream();
        ObjectOutputStream oos = new ObjectOutputStream(outputstream);
        oos.writeObject(pCommand);
        oos.flush();
        byte[] bytes = outputstream.toByteArray();
        oos.close();
```

```
79          clientSocket.getOutputStream().write(bytes);
80        }
81      }
82      catch (IOException pE)
83      {
84        throw new AditoIOException(pE, 16, 208, "type: " + pCommand.getType());
85      }
86    }
87
88    @Override
89    public boolean addListener(IRemoteLoggerListener pListener)
90    {
91      synchronized (loggerListenerList)
92      {
93        return loggerListenerList.add(pListener);
94      }
95    }
96
97    @Override
98    public boolean removeListener(@Nullable IRemoteLoggerListener pListener)
99    {
100     synchronized (loggerListenerList)
101     {
102       return loggerListenerList.remove(pListener);
103     }
104   }
105
106   /**
107    * Feuert, dass der RemoteLogger einen neuen CheckPoint gebracht hat
108    *
109    * @param pCheckPoint CheckPoint, den der RemoteLogger gebracht hat, nicht <tt>null</tt>!
110    */
111   private void _fireCheckPointReceived(@NotNull IRemoteLoggerCheckPoint pCheckPoint)
112   {
113     synchronized (loggerListenerList)
114     {
115       for (IRemoteLoggerListener currListener : loggerListenerList)
116         currListener.checkPointReceived(pCheckPoint);
117     }
118   }
119
120   /**
121    * Feuert, dass der RemoteLogger seinen Verbindungsstatus geänder hat
122    *
123    * @param pConnectedNow <tt>true</tt>, wenn der Logger jetzt mit dem am Server verbunden
124    *                      ist, bei <tt>false</tt> hat sich dieser getrennt
125    */
126   private void _fireConnectionStatusChanged(boolean pConnectedNow)
127   {
128     synchronized (loggerListenerList)
129     {
130       for (IRemoteLoggerListener currListener : loggerListenerList)
131         currListener.connectionStatusChanged(pConnectedNow);
132     }
133   }
134
135   /**
136    * Wird aufgerufen, wenn im ObjectStreamConsumer eine Fehlermeldung aufgetreten ist
137    *
138    * @param pException  Exception, die aufgetreten ist
139    * @return <tt>true</tt>, wenn aus dem Stream erneut ausgelesen werden soll
140    */
141   private boolean _handleException(@NotNull Exception pException)
142   {
143     CPH.checkPoint(pException, 16, 210, CPH.OK_DIALOG);
144     _fireConnectionStatusChanged(false);
145     return false;
146   }
147 }
```

11.1.11 Implementierungen des Interfaces „IRemoteLoggerListener“

Listener innerhalb des ADITO4-Managers zur Anzeige aufgetretener CheckPoints innerhalb der TreeTable:

```
/**
 * Listener um auf Ausgaben und Meldungen im RemoteLogger zu hören.
 * Fügt diesen dann in die TreeTable des ADTIO4-Managers ein
 */
private class _RemoteLoggerListener implements IRemoteLoggerListener
{
  @Override
  public void checkPointReceived(@NotNull IRemoteLoggerCheckPoint pCheckPoint)
  {
    treeTable.addCheckPoint(pCheckPoint);
  }

  @Override
  public void connectionStatusChanged(boolean pIsConnectedNow)
  {
    SwingUtilities.invokeLater(() -> connectionButton.setSelected(pIsConnectedNow));
  }
}
```

Listener innerhalb des IRemoteLoggerClientConnectionManagers, damit dieser aufgetretene CheckPoints an seine eigenen Listener verteilen kann.

```
/**
 * Listener der Nachrichten von der Connection
 * an alle eingehängten RemoteLogger verteilt
 */
private class _ControllingListener implements IRemoteLoggerListener
{
  @Override
  public void connectionStatusChanged(boolean pIsConnectedNow)
  {
    synchronized (listenerList)
    {
      listenerList.forEach(pLoggerListener ->
        pLoggerListener.connectionStatusChanged(pIsConnectedNow));
    }
  }

  @Override
  public void checkPointReceived(@NotNull IRemoteLoggerCheckPoint pCheckPoint)
  {
    synchronized (listenerList)
    {
      listenerList.forEach(pLoggerListener ->
        pLoggerListener.checkPointReceived(pCheckPoint));
    }
  }
}
```

Listener des JUnit-Tests der empfangene CheckPoints in der übergebenen Referenz ablegt.

```
/**
 * Hört darauf, wann CheckPoints empfangen werden.
 * Diese werden in der übergebenen Referenz gespeichert.
 * Alle Threads, die auf diese Referenz warten werden benachrichtigt
 */
private static class _RemoteListener implements IRemoteLoggerListener
{
  private final AtomicReference<IRemoteLoggerCheckPoint> refToSet;

  public _RemoteListener(AtomicReference<IRemoteLoggerCheckPoint> pRefToSet)
  {
    refToSet = pRefToSet;
  }

  @Override
  public void checkPointReceived(@NotNull IRemoteLoggerCheckPoint pCheckPoint)
  {
```

```
18      synchronized (refToSet)
19      {
20        refToSet.set(pCheckPoint);
21        refToSet.notifyAll();
22      }
23    }
24
25    @Override
26    public void connectionStatusChanged(boolean pIsConnectedNow)
27    {
28    }
29  }
```

11.1.12 Implementierung des Interfaces „IRemoteLoggerClientConnectionManager“

```
1   /**
2    * Implementierung eines RemoteLoggerClientConnectionManagers.
3    * Dieser dient als Zwischenstelle von außen auf die ClientConnection,
4    * damti diese bei Bedarf ausgewechselt werden kann
5    *
6    * @author W.Glanzer, 20.11.2015
7    */
8   public abstract class AbstractRemoteLoggerClientConnectionManager
9     implements IRemoteLoggerClientConnectionManager
10  {
11    private final List<IRemoteLoggerListener> listenerList = new ArrayList<>();
12    private final IRemoteLoggerListener listenerController = new _ControllingListener();
13    private IRemoteLoggerClientConnection currentConnection;
14
15    @Override
16    public void shutdown() throws AditoIOException
17    {
18      try
19      {
20        disconnect();
21        synchronized (listenerList)
22        {
23          listenerList.clear();
24        }
25      }
26      catch(Exception e)
27      {
28        throw new AditoIOException(e, 16, 203);
29      }
30    }
31
32    @Override
33    public void connect(Locale pLocale) throws AditoIOException
34    {
35      try
36      {
37        currentConnection = createConnection();
38        if(currentConnection != null)
39        {
40          currentConnection.addListener(listenerController);
41          currentConnection.connect();
42
43          // Sprache einstellen
44          currentConnection.sendCommand(new LanguageCommand(pLocale));
45        }
46      }
47      catch(Exception e)
48      {
49        throw new AditoIOException(e, 16, 204);
50      }
51    }
52
53    @Override
54    public void disconnect() throws AditoIOException
55    {
56      try
57      {
58        if (currentConnection != null)
59        {
```

```
60          currentConnection.close();
61          currentConnection.removeListener(listenerController);
62          currentConnection = null;
63        }
64      }
65      catch (Exception e)
66      {
67        throw new AditoIOException(e, 16, 205);
68      }
69    }
70
71    @Override
72    public void addListener(IRemoteLoggerListener pLoggerListener)
73    {
74      synchronized (listenerList)
75      {
76        listenerList.add(pLoggerListener);
77      }
78    }
79
80    @Override
81    public void removeListener(IRemoteLoggerListener pLoggerListener)
82    {
83      synchronized (listenerList)
84      {
85        listenerList.remove(pLoggerListener);
86      }
87    }
88
89    /**
90     * Erstellt die Connection
91     *
92     * @return Die Connection
93     */
94    @Nullable
95    protected abstract IRemoteLoggerClientConnection createConnection()
96      throws AditoException;
97
98    /**
99     * Listener der Nachrichten von der Connection
100    * an alle eingehängten RemoteLogger verteilt
101    */
102   private class _ControllingListener implements IRemoteLoggerListener
103   {
104     @Override
105     public void connectionStatusChanged(boolean pIsConnectedNow)
106     {
107       synchronized (listenerList)
108       {
109         listenerList.forEach(pLoggerListener ->
110           pLoggerListener.connectionStatusChanged(pIsConnectedNow));
111       }
112     }
113
114     @Override
115     public void checkPointReceived(@NotNull IRemoteLoggerCheckPoint pCheckPoint)
116     {
117       synchronized (listenerList)
118       {
119         listenerList.forEach(pLoggerListener ->
120           pLoggerListener.checkPointReceived(pCheckPoint));
121       }
122     }
123   }
124 }
```


© Korbinian Mifka

11.1.13 GUI-Implementierung

Hauptkomponente für die GUI des Remote-Logger-Clients im ADITO4-Manager

```java
/**
 * Panel des RemoteLogger-Clients im Manager
 *
 * @author W.Glanzer, 19.11.2015
 */
public class RemoteLoggerPanel extends AbstractDetailPanel
{
  private final RemoteLoggerTreeTable treeTable;
  private final RemoteLoggerDetailsPanel detailsPanel;
  private JToggleButton connectionButton;

  public RemoteLoggerPanel(RemoteLoggerNode pNode, SMPrefs pPrefs)
  {
    super(pPrefs);
    treeTable = new RemoteLoggerTreeTable();
    treeTable.setBorder(new CompoundBorder(new EmptyBorder(2, 0, 0, 0),
                                           treeTable.getBorder()));
    treeTable.getTreeTable().getSelectionModel()
      .addListSelectionListener(new _TreeTableSelectionListener());

    detailsPanel = new RemoteLoggerDetailsPanel();

    final AbstractFoldableSplitpane split = SplitpaneFactory
      .createSplitPane(AbstractFoldableSplitpane.TOP_TO_BOTTOM, treeTable,
                       detailsPanel, true, true);
    split.setFoldable(IFoldableSplitpane.SECOND_FOLDABLE);
    split.setBorder(null);
    IRemoteLoggerClientConnectionManager connectionManager = pNode.getConnectionManager();
    connectionManager.addListener(new _RemoteLoggerListener());
    add(_createToolBar(connectionManager), BorderLayout.NORTH);
    add(split, BorderLayout.CENTER);
    SwingUtilities.invokeLater(() -> {
      split.setDividerLocation((int) ((split.getPreferredSize().getHeight() / 4) * 1.5));
      split.revalidate();
      split.repaint();
    });
  }

  /**
   * Erstellt die Toolbar, zum Steuern des RemoteLogger-Clients
   *
   * @param pManager  ConnectionManager zum steuern des RL-Clients
   * @return Die Toolbar-Komponente
   */
  private JToolBar _createToolBar(IRemoteLoggerClientConnectionManager pManager)
  {
    JToolBar toolbar = new JToolBar();
    toolbar.setFloatable(false);

    connectionButton = new JToggleButton(Imageloader.getInstance().loadIcon(IIcons.LINK_16));
    connectionButton.setToolTipText(Translator.getString(16,
      IStaticResources.TEXT_CONNECT_TO_LOGGER, Locale.getDefault()));
    connectionButton.addActionListener(new _ConnectButtonListener(pManager));
    toolbar.add(connectionButton);

    JButton clearOutput = new JButton(Imageloader.getInstance().loadIcon(IIcons.TRASH_16));
    clearOutput.setToolTipText(Translator.getString(16,
      IStaticResources.TEXT_CLEAR_OUTPUT, Locale.getDefault()));
    clearOutput.addActionListener(pEvent -> treeTable.clear());
    toolbar.add(clearOutput);

    return toolbar;
  }

  @Override
  public TableSorter getSorter()
  {
    return null;
  }

  @Override
  public JXTable getTable()
  {
    return null;
  }
```

```
74    /**
75     * Implementierung eines ActionListeners der aufgerufen wird,
76     * wenn der connectionButton gedrückt wird
77     */
78    private class _ConnectButtonListener implements ActionListener
79    {
80      private IRemoteLoggerClientConnectionManager connectionManager;
81
82      public _ConnectButtonListener(IRemoteLoggerClientConnectionManager pConnectionManager)
83      {
84        connectionManager = pConnectionManager;
85      }
86
87      @Override
88      public void actionPerformed(ActionEvent e)
89      {
90        try
91        {
92          if(e.getSource() instanceof JToggleButton)
93          {
94            JToggleButton button = (JToggleButton) e.getSource();
95            boolean selected = button.isSelected();
96
97            // Wieder auf dem normalzustand setzen, da wir selbst das "selected"-Flag setzen!
98            button.setSelected(!selected);
99
100           if (selected)
101             connectionManager.connect(Locale.getDefault());
102           else
103             connectionManager.disconnect();
104         }
105       }
106       catch (Exception ex)
107       {
108         CPH.checkPoint(ex, 16, 209, CPH.OK_DIALOG);
109       }
110     }
111   }
112
113   /**
114    * Listener um auf Ausgaben und Meldungen im RemoteLogger zu hören
115    */
116   private class _RemoteLoggerListener implements IRemoteLoggerListener
117   {
118     @Override
119     public void checkPointReceived(@NotNull IRemoteLoggerCheckPoint pCheckPoint)
120     {
121       treeTable.addCheckPoint(pCheckPoint);
122     }
123
124     @Override
125     public void connectionStatusChanged(boolean pIsConnectedNow)
126     {
127       SwingUtilities.invokeLater(() -> connectionButton.setSelected(pIsConnectedNow));
128     }
129   }
130
131   /**
132    * Dieser Listener springt an, wenn sich
133    * die Selektion innerhalb der TreeTable ändert
134    */
135   private class _TreeTableSelectionListener implements ListSelectionListener
136   {
137     @Override
138     public void valueChanged(ListSelectionEvent e)
139     {
140       int selectedRow = treeTable.getTreeTable().getSelectedRow();
141       if(selectedRow <= -1)
142         detailsPanel.setContent(null);
143       else
144       {
145         Object row = treeTable.getTreeTable().getModel().getValueAt(selectedRow, 0);
146         if (row instanceof DefaultMutableTreeNode)
147         {
148           Object uo = ((DefaultMutableTreeNode) row).getUserObject()
149           if (uo instanceof IRemoteLoggerCheckPoint)
150             detailsPanel.setContent((IRemoteLoggerCheckPoint) uo);
151         }
152       }
153     }
154   }
155 }
```


© Korbinian Mifka

Detail-Panel unterhalb der TreeTable im ADITO4-Manager

```
/**
 * Stellt eine TextArea dar, die selbst die
 * Detailansicht eines RemoteLoggerCheckPoints
 * darstellen kann. Zeigt bspw den StackTrace eines
 * o.g. CheckPoints an
 *
 * @author W.Glanzer, 27.11.2015
 */
class RemoteLoggerDetailsPanel extends ScrollTextArea
{

  public RemoteLoggerDetailsPanel()
  {
    setBorder(null);
    setOpaque(true);
    setBackground(LfUtil.get().getGuiColors().getTaskpaneContainerColorBright());

    getTextArea().setEditable(false);
    getTextArea().setFocusable(false);
  }

  /**
   * Gibt an, welcher CheckPoint genauer angezeigt werden soll, oder <tt>null</tt>,
   * wenn die TextArea geleert werden soll
   *
   * @param pCheckPoint  CheckPoint der angezeigt werden soll, oder <tt>null</tt>
   */
  public void setContent(@Nullable IRemoteLoggerCheckPoint pCheckPoint)
  {
    String t = _toDisplayString(pCheckPoint != null ? pCheckPoint.getDetails() : null)
    getTextArea().setText(t);
    SwingUtilities.invokeLater(() -> getVerticalScrollBar().setValue(0));
  }

  /**
   * Wandelt ein StringArray in einen normalen String um
   *
   * @param pDetails  Details eines CheckPoints
   * @return Ein String, der das Detail-Array repräsentiert, nicht <tt>null</tt>
   */
  @NotNull
  private String _toDisplayString(String[] pDetails)
  {
    if(pDetails == null)
      return "";

    StringBuilder builder = new StringBuilder();
    for (String currDetail : pDetails)
      builder.append(currDetail).append('\n');
    return builder.toString();
  }
}
```

TreeTable, in der die CheckPoints angezeigt werden

```
/**
 * Anzeige der CheckPoints in einer TreeTable
 *
 * @author W.Glanzer, 23.11.2015
 */
class RemoteLoggerTreeTable extends JPanel
{
  private final SplitsTreeTable treeTable;
  private final RemoteLoggerTableModel model = new RemoteLoggerTableModel();

  public RemoteLoggerTreeTable()
  {
    setLayout(new BorderLayout());
    treeTable = new SplitsTreeTable(DefaultOutlineModel.createOutlineModel(model, model,
     true, Translator.getString(16, IStaticResources.TEXT_TIMESTAMP, Locale.getDefault())));

    treeTable.setRootVisible(false);
    treeTable.setRenderDataProvider(new RemoteLoggerTreeDataProvider());
    treeTable.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
    treeTable.setDefaultRenderer(IRemoteLoggerCheckPoint.class,
      new RemoteLoggerTreeTableCheckPointRenderer());
```

```
22      for (TableColumn currColumn : treeTable.getColumns())
23      {
24        int currIndex = treeTable.getColumnModel().getColumnIndex(currColumn.getIdentifier());
25        if(currIndex == 0)
26          currColumn.setPreferredWidth(150);
27        else if(currIndex == 1)
28          currColumn.setPreferredWidth(90);
29        else
30          currColumn.setPreferredWidth(300);
31      }
32
33      treeTable.addFocusListener(new FocusAdapter()
34      {
35        @Override
36        public void focusLost(FocusEvent e)
37        {
38          // Beim wechsel des Fokus ist ein behalten der Selektion verwirrend
39          treeTable.clearSelection();
40        }
41      });
42
43      SwingUtilities.invokeLater(() -> {
44        add(treeTable.getTableHeader(), BorderLayout.NORTH);
45        add(new AutoScrollDownScrollPane(treeTable, true), BorderLayout.CENTER);
46      });
47    }
48
49    /**
50     * Fügt dem Model einen CheckPoint hinzu
51     *
52     * @param pCheckPoint CheckPoint, der hinzugefügt werden soll, nicht
53     *                    <tt>null</tt>, da das keinen Sinn machen würde!
54     */
55    public void addCheckPoint(@NotNull IRemoteLoggerCheckPoint pCheckPoint)
56    {
57      // Alle derzeit expandierten Pfade merken
58      TreePathSupport tpSupport = treeTable.getOutlineModel().getTreePathSupport();
59      TreePath[] allExpandeds = tpSupport.getExpandedDescendants(new TreePath(model.getRoot()));
60
61      // derzeit selektierte Zeile speichern
62      int selectedRow = treeTable.getSelectedRow();
63
64      // Dem Model einen neuen CheckPoint hinzufügen
65      model.addCheckPoint(pCheckPoint);
66
67      // expandierte Pfade wieder expandieren
68      for (TreePath currPath : allExpandeds)
69        tpSupport.expandPath(currPath);
70
71      // selektiert die davor gespeicherte Zeile wieder
72      if(selectedRow > -1)
73        treeTable.setRowSelectionInterval(selectedRow, selectedRow);
74
75      SwingUtilities.invokeLater(() -> {
76        revalidate();
77        repaint();
78      });
79    }
80
81    /**
82     * Leert das Ausgabefenster
83     */
84    public void clear()
85    {
86      model.clear();
87
88      SwingUtilities.invokeLater(() -> {
89        treeTable.revalidate();
90        treeTable.repaint();
91        revalidate();
92        repaint();
93      });
94    }
95
96    /**
97     * @return Die konkrete Instanz der dahinterliegenden TreeTable
98     */
99    public SplitsTreeTable getTreeTable()
100   {
101     return treeTable;
102   }
103 }
```


© Korbinian Mifka

Model, das die Daten für die Tabelle der o.g. TreeTable liefert

```
/**
 * Model der RemoteLoggerTreeTable, zur Visualisierung von CheckPoints.
 * Stellt das Model der rechts-liegenden Tabelle dar (2. Teil der TreeTable)
 *
 * @see RemoteLoggerTreeDataProvider
 * @see RemoteLoggerTreeTable
 * @author W.Glanzer, 24.11.2015
 */
class RemoteLoggerTableModel extends DefaultTreeModel implements RowModel
{
  private final DefaultMutableTreeNode rootNode = new DefaultMutableTreeNode();

  public RemoteLoggerTableModel()
  {
    super(null);
    setRoot(rootNode);
  }

  /**
   * Fügt einen CheckPoint in das Model ein und lädt das Model neu
   *
   * @param pCheckPoint CheckPoint, der hinzugefügt werden soll, nicht <tt>null</tt>!
   */
  public void addCheckPoint(@NotNull IRemoteLoggerCheckPoint pCheckPoint)
  {
    rootNode.add(_toNode(pCheckPoint));
    reload(rootNode);
  }

  @Override
  public int getColumnCount()
  {
    return 2; //+1 Tree-Spalte
  }

  @Override
  public Object getValueFor(Object node, int column)
  {
    return node;
  }

  @Override
  public Class getColumnClass(int column)
  {
    return IRemoteLoggerCheckPoint.class;
  }

  @Override
  public boolean isCellEditable(Object node, int column)
  {
    // Generell keine Editiermöglichkeit erlaubt!
    return false;
  }

  @Override
  public void setValueFor(Object node, int column, Object value)
  {
    // Generell keine Editiermöglichkeit erlaubt!
  }

  @Override
  public String getColumnName(int column)
  {
    switch(column)
    {
      case 0:
        return Translator.getString(16, IStaticResources.TEXT_IDENTIFICATION,
                                    Locale.getDefault());

      case 1:
        return Translator.getString(16, IStaticResources.TEXT_MESSAGE,
                                    Locale.getDefault());

      default:
        return null;
    }
  }
```

```
78    /**
79     * Leert alle Einträge in der RootNode --> Keine LogAusgaben mehr angezeigt
80     */
81    public void clear()
82    {
83      rootNode.removeAllChildren();
84      reload(rootNode);
85    }
86
87    /**
88     * Wandelt einen CheckPoint in eine darstellbare TreeNode um
89     *
90     * @param pCheckPoint  CheckPoint, der umgewandelt werden soll
91     * @return TreeNode, die angezeigt werden kann und die Baumstruktur darstellt
92     */
93    private MutableTreeNode _toNode(IRemoteLoggerCheckPoint pCheckPoint)
94    {
95      DefaultMutableTreeNode node = new DefaultMutableTreeNode(pCheckPoint);
96      if(pCheckPoint.getCause() != null)
97        node.add(_toNode(pCheckPoint.getCause()));
98      return node;
99    }
100
101 }
```

Modell, das die Daten für den Baum der o.g. TreeTable liefert

```
1   /**
2    * Stellt die Daten für den Tree der TreeTable bereit
3    *
4    * @author W.Glanzer, 24.11.2015
5    */
6   class RemoteLoggerTreeDataProvider implements RenderDataProvider
7   {
8
9     @Override
10    public String getDisplayName(Object pNode)
11    {
12      IRemoteLoggerCheckPoint checkPoint = _toCP(pNode);
13      if(checkPoint != null)
14        return DateUtility.dateToISO8601(new Date(checkPoint.getTime()), DateUtility.UTC);
15
16      return String.valueOf(pNode);
17    }
18
19    @Override
20    public String getTooltipText(Object pNode)
21    {
22      return null;
23    }
24
25    @Override
26    public boolean isHtmlDisplayName(Object pNode)
27    {
28      return false;
29    }
30
31    @Override
32    public Color getBackground(Object pNode)
33    {
34      return null;
35    }
36
37    @Override
38    public Color getForeground(Object pNode)
39    {
40      IRemoteLoggerCheckPoint checkPoint = _toCP(pNode);
41      if(checkPoint != null)
42      {
43        IGuiColors colors = LfUtil.get().getGuiColors();
44        return CheckPointUtility.getColor(checkPoint.getPriority(), colors.getErrorColor(),
45          colors.getWarningColor(), colors.getInfoColor());
46      }
47
48      return null;
49    }
```

```
@Override
  public Icon getIcon(Object pNode)
  {
    // Kein Icon, auch kein standardicon!
    return new EmptyIcon(0, 0);
  }

  /**
   * Wandelt ein Object in einen RemoteLoggerCheckPoint um, wenn dies funktioniert
   *
   * @param pNode  Object, das umgewandelt werden soll
   * @return Einen RemoteLoggerCheckPoint, oder <tt>null</tt>, wenn es nicht möglich war
   */
  private IRemoteLoggerCheckPoint _toCP(Object pNode)
  {
    if(pNode instanceof DefaultMutableTreeNode)
    {
      Object obj = ((DefaultMutableTreeNode) pNode).getUserObject();
      if (obj instanceof IRemoteLoggerCheckPoint)
        return (IRemoteLoggerCheckPoint) obj;
    }

    return null;
  }
}
```

Rendert die CheckPoints innerhalb der o.g. TreeTable

```
/**
 * Rendert einen CheckPoint in der TreeTable (rechten Tabelle)
 *
 * @author W.Glanzer, 24.11.2015
 */
class RemoteLoggerTreeTableCheckPointRenderer extends DefaultOutlineCellRenderer
{

  @Override
  public Component getTableCellRendererComponent(JTable pTable, Object pValue,
    boolean pIsSelected, boolean pHasFocus, int pRow, int pColumn)
  {
    super.getTableCellRendererComponent(pTable, pValue, pIsSelected, false, pRow, pColumn);
    if(pValue instanceof DefaultMutableTreeNode &&
      ((DefaultMutableTreeNode) pValue).getUserObject() instanceof IRemoteLoggerCheckPoint)
    {
      DefaultMutableTreeNode dmtn = (DefaultMutableTreeNode) pValue
      IRemoteLoggerCheckPoint checkPoint = (IRemoteLoggerCheckPoint) dmtn.getUserObject();

      // Sonst würde die weiße Vordergrundfarbe überschrieben
      // --> Schlecht leesbar, wenn selektiert
      if(!pIsSelected)
      {
        IGuiColors colors = LfUtil.get().getGuiColors();
        setForeground(CheckPointUtility.getColor(checkPoint.getPriority(),
          colors.getErrorColor(), colors.getWarningColor(), colors.getInfoColor()));
      }

      switch (pColumn)
      {
        case 1:
          setText(_getID(checkPoint));
          break;

        case 2:
          setText(_getMessage(checkPoint));
          break;
      }
    }

    return this;
  }
```

```
/**
   * Liefert den Identifier des CheckPoints
   *
   * @param pCheckPoint  CheckPoint, dessen ID gesucht ist
   * @return ID als String
   */
  private String _getID(IRemoteLoggerCheckPoint pCheckPoint)
  {
    return CheckPointUtility.getErrorCode(pCheckPoint.getKind(), pCheckPoint.getModule(),
      pCheckPoint.getPriority(), pCheckPoint.getID(), pCheckPoint.getProgram());
  }

  /**
   * Liefert die zusammengesetzte Nachricht des CheckPoints.
   * Diese setzt sich aus der originalen Message und der Details zusammen
   *
   * @param pCheckPoint  CheckPoint, dessen Nachricht ausgelesen werden soll
   * @return Message + Details als einzelner String
   */
  private String _getMessage(IRemoteLoggerCheckPoint pCheckPoint)
  {
    return pCheckPoint.getMessage();
  }

}
```

11.1.14 Test mit JUnit

```
/**
 * Testet den RemoteLogger auf Funktionsfähigkeit
 *
 * @author W.Glanzer, 04.12.2015
 */
public class Test_RemoteLogger
{
  private RemoteLogger logger;
  private final AtomicReference<IRemoteLoggerCheckPoint> lastCheckPointGER =
    new AtomicReference<>();
  private final AtomicReference<IRemoteLoggerCheckPoint> lastCheckPointENG =
    new AtomicReference<>();

  /**
   * Initialisiert den Test.
   * Baut den RemoteLogger (localhost, 7733) und den CheckPointHandler auf, damit
   * diese verwendet werden können.
   */
  @Before
  public void init() throws Exception
  {
    logger = new RemoteLogger('Z', "localhost", 7733, _DummyFacade::new);
    CheckPointHandler cph = CPH.init('Z', logger);
    Assert.assertNotNull(cph);
  }

  /**
   * Testet, ob der Remote-Logger einwandfrei funktioniert.
   * Dafür werden zwei Remote-Logger-Clients erzeugt. Diesen werden zwei getrennte Listener
   * eingehangen, die ihre Werte in die jeweils übergebene Referenz speichern.
   * Auf diese Werte wird in dieser Methode gewartet und anschließend verglichen,
   * ob die Werte in deutscher und in englischer Sprache vorhanden und korrekt sind.
   */
  @Test
  public void test_communication() throws Exception
  {
    IRemoteLoggerClientConnectionManager manager = _startClient(Locale.GERMAN);
    IRemoteLoggerClientConnectionManager manager2 = _startClient(Locale.ENGLISH);
    Assert.assertNotNull(manager);
    Assert.assertNotNull(manager2);

    manager.addListener(new _RemoteListener(lastCheckPointGER));
    manager2.addListener(new _RemoteListener(lastCheckPointENG));

    CPH.checkPoint(99, 9999);
    Assert.assertEquals(_getNextCheckPoint(lastCheckPointGER).getID(), 9999);
    Assert.assertEquals(_getNextCheckPoint(lastCheckPointENG).getID(), 9999);
```

```
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(), "Interner Fehler. Bitte kontaktieren Sie
53                          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());
59
60      cp = _getNextCheckPoint(lastCheckPointENG);
61      Assert.assertEquals(cp.getModule(), 0);
62      Assert.assertEquals(cp.getID(), 1);
63      Assert.assertEquals(cp.getMessage(), "Internal error. Please contact administrator.");
64      Assert.assertEquals(cp.getProgram(), 'Z');
65      Assert.assertEquals(cp.getKind(), 'B');
66      Assert.assertEquals(cp.getPriority(), 'D');
67      Assert.assertTrue(cp.getTime() > 0);
68      Assert.assertTrue(cp.getTime() <= System.currentTimeMillis());
69    }
70
71    /**
72     * Beendet den Test und fährt den Logger herunter.
73     * Dieser gibt dann seinen Socket wieder frei
74     */
75    @After
76    public void shutdown() throws Exception
77    {
78      logger.destroy();
79      Assert.assertTrue(true);
80    }
81
82    /**
83     * Blockiert so lange, bis ein neuer CheckPoint eingetroffen ist
84     *
85     * @param pRefToWaitOn Referenz, in der ein neuer CheckPoint gespeichert wird
86     * @return Der empfangene CheckPoint
87     */
88    @NotNull
89    private IRemoteLoggerCheckPoint _getNextCheckPoint(final
90      AtomicReference<IRemoteLoggerCheckPoint> pRefToWaitOn)
91    {
92      if(pRefToWaitOn.get() == null)
93      {
94        synchronized (pRefToWaitOn)
95        {
96          try
97          {
98            pRefToWaitOn.wait();
99          }
100         catch (InterruptedException ignored)
101         {
102         }
103       }
104     }
105
106     IRemoteLoggerCheckPoint cp = pRefToWaitOn.getAndSet(null);
107     Assert.assertNotNull(cp);
108     return cp;
109   }
110
111   /**
112    * Startet einen neuen ConnectionManager, verbindet sich
113    * mit dem Server mit einer angegebenen Lokale und gibt diesen
114    * Manager dann zurück
115    *
116    * @return Manager-Instanz der Verbindung
117    */
118   @Nullable
119   private static IRemoteLoggerClientConnectionManager _startClient(Locale pLocale)
120     throws AditoIOException
121   {
122     _ConnectionManager manager = new _ConnectionManager();
123     manager.connect(pLocale);
124     return manager;
125   }
```

```
126   /**
127    * Implementierung einer LoginFacade, die die
128    * LoginInformationen "USER"-"PASS" erwartet.
129    * Dadurch lässt sich die Login-Funktionalität prüfen!
130    */
131   private static class _DummyFacade implements IRemoteLoggerLoginFacade
132   {
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     }
140   }
141
142   /**
143    * Implementiert einen ConnectionManager, der sich auf
144    * den Remote-Logger dieses Testes verbindet
145    */
146   private static class _ConnectionManager
147     extends AbstractRemoteLoggerClientConnectionManager
148   {
149     @Nullable
150     @Override
151     protected IRemoteLoggerClientConnection createConnection() throws AditoException
152     {
153       String[] login = new String[]{"USER", "PASS"};
154       return new RemoteLoggerClientConnection("localhost", 7733, login);
155     }
156   }
157
158   /**
159    * Hört darauf, wann CheckPoints empfangen werden.
160    * Diese werden in der übergebenen Referenz gespeichert.
161    * Alle Threads, die auf diese Referenz warten werden benachrichtigt
162    */
163   private static class _RemoteListener implements IRemoteLoggerListener
164   {
165     private final AtomicReference<IRemoteLoggerCheckPoint> refToSet;
166
167     public _RemoteListener(AtomicReference<IRemoteLoggerCheckPoint> pRefToSet)
168     {
169       refToSet = pRefToSet;
170     }
171
172     @Override
173     public void checkPointReceived(@NotNull IRemoteLoggerCheckPoint pCheckPoint)
174     {
175       synchronized (refToSet)
176       {
177         refToSet.set(pCheckPoint);
178         refToSet.notifyAll();
179       }
180     }
181
182     @Override
183     public void connectionStatusChanged(boolean pIsConnectedNow)
184     {
185     }
186   }
187 }
```


© Korbinian Mifka