JOHANNES KEPLER UNIVERSITÄT LINZ | JKU

TNF

Technisch-Naturwissenschaftliche Fakultät

Webkonsole für die Plugin-Plattform Plux.NET

BACHELORARBEIT
(Projektpraktikum)

zur Erlangung des akademischen Grades

Bachelor of Science

im Bachelorstudium

INFORMATIK

Eingereicht von:
Benjamin Rosenberger, 0657377

Angefertigt am:
Institut für Systemsoftware

Beurteilung:
Mag. Dr. Reinhard Wolfinger

Wels, März 2012

Inhaltsverzeichnis

1. Einleitung

Ziel dieses Projekts ist es, mit dem Google Web Toolkit (GWT) [2] eine Webkonsole für Plux [14] zu programmieren. Die Webkonsole ist ein verteiltes Programm das auf heterogenen Technologien basiert. Verteilt deshalb, weil das (zu programmierende) Kommandozeilenfenster auf dem Webclient läuft, der bereits vorhandene Befehlsinterpreter hingegen auf dem Webserver läuft. Heterogen deshalb, weil das Kommandozeilenfenster in GWT (das ist JavaScript) implementiert ist, während der Befehlsinterpreter in .NET implementiert ist. Die Kommunikation zwischen Client und Server soll auf JSON [12] und WebSockets [6] basieren.

1.1. Überblick

Kapitel 2 erläutert die technischen Grundlagen für dieses Projekt.

Das 3. Kapitel widmet sich der Programmstruktur und zeigt wie das Problem der Verbindung der Server- und Clientkomponenten gelöst wurde.

Im 4. Kapitel wird gezeigt wie einzelne Teilprobleme gelöst wurden.

Kapitel 5 beurteilt die Ergebnisse dieser Arbeit.

2. Grundlagen

Dieses Projekt nutzt folgende Technologien: Plux-Plugin-Framework, WebSockets, JSON und das Google-Web-Toolkit-Framework (GWT).

Plux stellt die Serverkomponente bereit, die es Plugins ermöglicht das ausführbare Programm zu konfigurieren und zu erweitern.

WebSockets verbinden Plux-Plugin und den Web-Client bidirektional.

GWT wird genutzt um den Web-Client als Java-Anwendung zu entwickeln, die mittel Compiler in eine JavaScript Anwendung konvertiert wird.

2.1.Plux-Plugin-Framework

Plux.NET [14] ist eine Bibliothek zur Erstellung von Plugin-Systemen unter .NET und wurde vom Christian Doppler Laboratory for Automated Software Engineering [19] und dem Institut für Systemsoftware an der Johannes Keppler Universität Linz [20] entwickelt.

Die Kernkomponente zum konfigurieren der Anwendung ist der Plux-Interpreter. Dieser bietet die Möglichkeit zur Laufzeit Plugins hinzuzufügen oder zu entfernen. In der Plux-Terminologie wird ein Plugin (also ein Erweiterung) als Extension bezeichnet, die erweiterbare Stelle in der Anwendung wird als Slot bezeichnet. Über Parametrisierung der Extensions und der Slots kann Plux die entsprechenden Plugins mit der Anwendung verbinden („pluggen“).

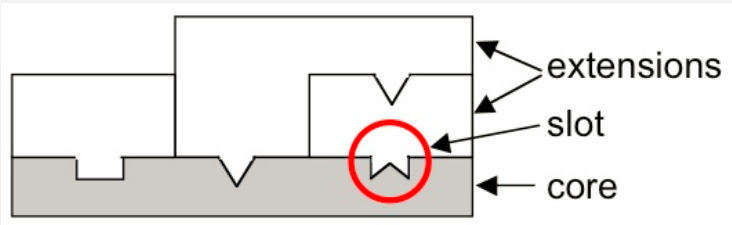

Abbildung 1: Über Steckplätze erweiterbares Plux-Kernsystem [14]

2.2.JSON

JSON (JavaScript Object Notation) [12] ist ein für Mensch und Maschine lesbares Format (siehe Beispiel 1) zum Kodieren von Objekten. Durch zahlreiche Unterstützung durch viele Programmiersprachen, darunter auch C# und JavaScript, dient es als Kommunikationsprotokoll zwischen Server und Client zum Übertragen von Objekten.

```
Klassendefinition:
class SendObject {
   String[] values;
   Integer count;
}

Objekt:
toSend = new SendObject(2, “a“, “b“)

JSON:
{“values“:[“a“,“b“],“count“:2}
```

Beispiel 1: Klassen-, Objektdefintion in Java und Objektrepräsentation in JSON

2.3. WebSockets

WebSockets ermöglichen eine bidirektionale Verbindung zwischen Server und Client. Sie wurden mit dem HTML5 Standard vorgestellt, der die Spezifikation der WebSockets in API und Protokoll unterteilt. Während die API bereits festgelegt ist befindet sich das Protokoll noch in der Entwicklungsphase. Der daraus entstandene Standard (RFC 6455) soll voraussichtlich Anfang April 2012 in Kraft treten.

WebSocket-Verbindungen werden über einen HTTP-Request initialisiert und können in Folge wie eine TCP-Socket-Verbindung verwendet werden. Im Gegensatz dazu sind herkömmliche HTTP-Requests und WebService-Aufrufe für den Webserver Stateless, da keine Informationen zu vorangegangenen Aufrufen zu einem Client vorhanden sind. Des Weiteren können auch keine Antworten vom Server an den Client gesendet werden, ohne dass eine Anfrage des Clients vorliegt.

Die Funktionalität, dass die Serverkomponente ohne vorherige Anfrage Daten an den Client schicken kann, war ausschlaggebend für die Verwendung von WebSockets in dieser Arbeit.

Die WebSocket API

Die WebSocket API [9] wurde vom W3C genormt und ist in Beispiel 2 auszugsweise mit den wichtigsten Funktionen dargestellt.

```
[Constructor(DOMString url, optional DOMString protocols),
 Constructor(DOMString url, optional DOMString[] protocols)]
interface WebSocket : EventTarget {
  readonly attribute DOMString url;

  readonly attribute unsigned short readyState;

  // networking
  [TreatNonCallableAsNull] attribute Function? onopen;
  [TreatNonCallableAsNull] attribute Function? onerror;
  [TreatNonCallableAsNull] attribute Function? onclose;
  void close([Clamp] optional unsigned short code, optional DOMString reason);

  // messaging
  [TreatNonCallableAsNull] attribute Function? onmessage;
           attribute DOMString binaryType;
  void send(DOMString data);
  void send(ArrayBuffer data);
  void send(Blob data);
};
```

Beispiel 2: Auszug WebSocket-Api Definition [9]

Wie in Beispiel 2 dargestellt besitzt ein WebSocket einen Konstruktor mit URL und optionalem Protokoll auf dem die Verbindung basiert. Dies kann ein eigens definierter Protokoll-Identifier sein, der mit dem Server vereinbart wird, und dem im Anschluss die Kommunikation unterliegt.

Weiters hat jeder WebSocket vier verschiedene Status in denen unterschiedlichen Aktionen möglich sind. Diese sind in Abbildung 2 ersichtlich.

Die Methoden close() und send() können benutzt werden um aktiv mit dem WebSocket Aktionen durchzuführen. Im Gegensatz dazu bieten die Methoden onopen(), onerror(), onclose() und onmessage() asynchron die Möglichkeit auf die entsprechenden Ereignisse des WebSockets zu reagieren.

Beispiel 3 zeigt ein kleines Beispiel das die WebSocket-API in JavaScript verwendet. Es wird ein WebSocket aufgebaut und sobald dieser verbunden ist wird eine Nachricht an den Server gesandt und die Verbindung wieder abgebrochen.

```
var websocket = new WebSocket("ws://websocket.server.url");

websocket.onopen = function() {
   websocket.send("socket successfully opened");
   websocket.close();
}
```

Beispiel 3: WebSocket-Api Verwendung mit JavaScript

Das WebSocket Protokoll

Das WebSocket Protokoll [7] dient zur Kommunikation zwischen Server und Client. Clientseitig wird dies durch den Browser abgewickelt, sodass nur noch der Zugriff mittels API nötig ist. Serverseitig muss oft in die vorhandenen Funktionen eingegriffen werden, da es für WebSockets noch keinen offiziellen Standard gibt und existierende OpenSource Lösungen oft noch nicht angepasst wurden.

Der Ablauf zum Senden von Daten über WebSockets erfolgt im fehlerfreien Fall immer in den Schritten die in Abbildung 2 zu sehen ist:

WebSocket	State	Action
new WebSocket(url)	opening	Opening Handshake
ws.onOpen(OpenEvent)	open	
ws.send(data)		Framing of Data, sending
ws.onMessage(message)	open	Framing of Data, receiving
ws.close()	closing	Closing Handshake
ws.onClose(CloseEvent)	closed	

Abbildung 2: Websocket-Workflow

Im Opening Handshake wird ausgemacht welches Protokoll für die Kommunikation genützt wird, der Sicherheitsschlüssel ausgetauscht und in Folge die bidirektionale Verbindung aufgebaut.

Der in Beispiel 4 gezeigte Request wurde mit WebSocket-Version 8 (draft-ietf-hybi-thewebsocketprotocol-10) erstellt. Die Versionsnummer wird bis zur Fertigstellung des Standards noch auf Version 13 ansteigen. Der Client teilt in diesem Beispiel dem Server mittels Protokoll Header mit, dass dieser das "chat" oder das "superchat"-Protokoll für die Nachrichtenübertragung nutzen kann.

```
GET /chat HTTP/1.1
Host: server.example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
Sec-WebSocket-Origin: http://example.com
Sec-WebSocket-Protocol: chat, superchat
Sec-WebSocket-Version: 8
```

Beispiel 4: WebSocket Request [7]

Der Server antwortet dann mit in Beispiel 5 gezeigter Response. Das WebSocket-Accept ergibt sich aus der Kombination des zuvor eingelesenen WebSocket-Key und einer im Draft festgesetzen Konstante (258EAFA5-E914-47DA-95CA-C5AB0DC85B11) die anschließend mittels SHA1 gehasht wird

und base64 kodiert wird. Der Server teilt weiters dem Client mit, dass er die Datenübertragung mittels "chat"-Protokoll führen möchte.

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocketConnection: Upgr ade
Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=
Sec-WebSocket-Protocol: chat
```

Beispiel 5: WebSocket Response [7]

Der Closing Handshake besteht aus einem simplen ClosingFrame, der aus einem OpCode und Grund des Schließens besteht.

WebSockets ermöglichen auch ein Framing (Abbildung 3) der übertragenen Daten, welches unterschiedlich eingesetzt werden kann. Etliche Kommunikationen benötigen nur einen einzigen Frame, welcher wie der Schlussframe einer Kette mit dem FIN-Bit=1 gekennzeichnet ist. Die reservierten Bytes 4 bis 7 (OpCode) kennzeichnen jeden Frame für dessen Verwendung und Inhalt. Aktuell sind Bitkombinationen für folgende Funktionalitäten definiert: „Weiterführender Frame“, „Textdaten“, „Binärdaten“, „Ping“, „Pong“ und „Schließen“. Alle weiteren möglichen Bitkombinationen sind für zukünftige Verwendung reserviert.

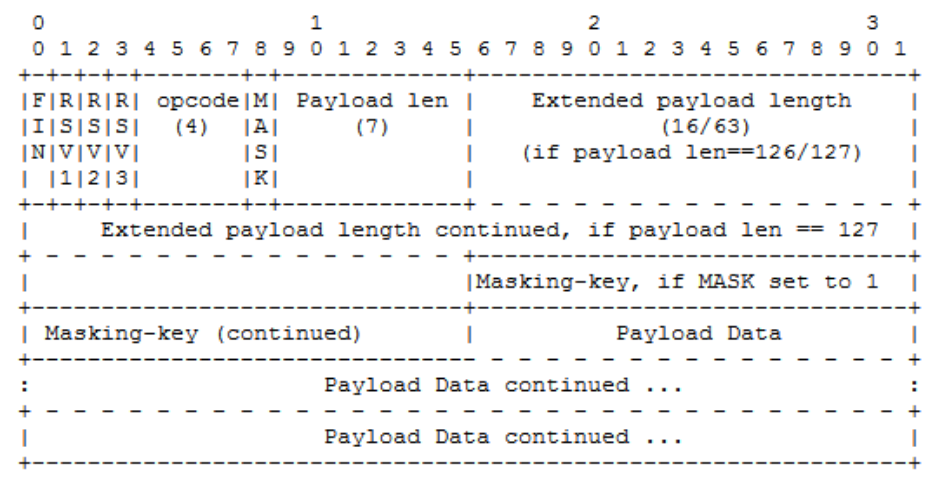

Abbildung 3: DataFraming bei WebSockets [7]

Aktuelle Browserkompatibilität

Zurzeit unterstützen nur wenige Browser den WebSocket Draft. Draft-10 wird von Firefox 7, Chrome 14 und InternetExplorer 10 Developer Preview umgesetzt. Die Webkonsole, die in dieser Arbeit implementiert wurde ist somit nur mit oben genannten Browsern kompatibel.

Obwohl Firefox 7 draft-10 unterstützt, werden nicht alle Features angeboten. Dies betrifft vor allem Binärnachrichten wie Abbildung 4 zeigt.

9 Limits/Performance	"WebSocket Snapshot/2011-10-27"		AutobahnClient/0.4.3		Chrome/15.0.874.106		Chrome/17.0.919.0		Firefox/10.0a(20111028)	Firefox/7.0.(20100101)	Firefox/8.(20100101)	Firefox/9.0a(20111027
9.2 Binary Message (increasing size)												
Case 9.2.1	Pass 273 ms	1000	Pass 60 ms	1000	Pass 46 ms	None	Pass 46 ms	None	Missing	Missing	Missing	Missing
Case 9.2.2	Pass 159 ms	1000	Pass 222 ms	1000	Pass 143 ms	None	Pass 143 ms	None	Missing	Missing	Missing	Missing
Case 9.2.3	Pass 808 ms	1000	Pass 894 ms	1000	Pass 884 ms	None	Pass 895 ms	None	Missing	Missing	Missing	Missing
Case 9.2.4	Pass 2588 ms	1000	Pass 3941 ms	1000	Pass 3539 ms	None	Pass 3720 ms	None	Missing	Missing	Missing	Missing
Case 9.2.5	Pass 4779 ms	1000	Pass 7890 ms	1000	Pass 11029 ms	None	Pass 10669 ms	None	Missing	Missing	Missing	Missing
Case 9.2.6	Pass 9799 ms	1000	Pass 14406 ms	1000	Pass 16500 ms	None	Pass 15699 ms	None	Missing	Missing	Missing	Missing

Abbildung 4: Testbeispiele zur Browserkompatibilität [8]

Zusätzlich ist der WebSocket in Firefox (und damit allen Gecko basierenden Browsern) nur mit Präfix „Moz“ erzeugbar, was eine Spezialbehandlung des JavaScript-Codes (siehe 4.4) nötig macht.

2.4.Google-Web-Toolkit (GWT)

Mit dem Google-Web-Toolkit [2] kann eine Webanwendung in Java programmiert werden, die zur Laufzeit im Browser als JavaScript-Anwendung ausgeführt wird. GWT übersetzt dazu die Anwendung in hochoptimierte browserspezifische JavaScript-Anwendungen (eine Anwendung für jede Browserfamilie).

GWT bietet etliche Vorteile die für dieses Projekt genutzt werden können:

- Darunter fällt das Programmieren in der high level language Java und dem damit verbundenen Programmierprinzipien wie Objektorientierung
- Viele Standardklassen wurden von Google für GWT gemappt und können somit in einer Webanwendung verwendet werden.
- Kapsellung von JavaScript-Funktionalität in GWT-Java-Klassen
- Templating von UI-Komponenten

GWT ist für eine ganzheitliche Client-Server Anwendung gedacht, da von GWT umfangreiche Methoden zur Serverkommunikation mittels RPC mitgeliefert werden. Dies ist für dieses Projekt jedoch nicht relevant und wird daher nicht weiter erläutert. Im Folgenden wird ausschließlich auf die Implementierung einer Client-Webanwendung im Browser eingegangen.

GWT unterteilt die erstellten Anwendungen in Module die von einem EntryPoint.onModuleLoad() (im Vergleich zu einer main-Methode in Standard Java-Programmen) gestartet wird. Im einfachsten Fall kann hier der gesamte Code der Anwendung zusammengefasst und programmiert werden. Um jedoch Wiederverwendbarkeit, Trennung der UI, Daten und Logik zu erreichen wird im Weiteren kurz das GWT-MVP (Modell-View-Presenter) erläutert.

Der Moduleinstiegspunkt instanziiert alle nötigen Klassen und verknüpft diese. Am Ende wird dann die Einstiegsseite des GWT-Projektes aufgerufen.

Die unterschiedlichen Komponenten die hier verwendet werden sind:

- Views und ClientFactory [5]
  Views (Beispiel 6) sind die eigentlichen Webseiten einer GWT-Anwendung. Sie werden mittels UI-Templates konfiguriert und mit einem UiBinder mit der View-Logik verknüpft. Die View-Logik reagiert auf Interaktionen des Nutzers auf Ui-Elemente (mittels @UiHandler

annotierte Methoden) und soll die Aktionen an eine Activity weiterdelegieren, die dann jegliche weitere Businesslogik ausführt und das Resultat wieder an die View weitergibt. Abgebildet wird die View als Interface, die alle Aktionen der View auflistet (z.B. setDtosToShow(Dto[] dtos)) und einem Subinterface, das die Aktionen auf die entsprechende Activity definiert (z.B. setDtoClicked(Long dtoId)). Dieses Interface wird auch als Presenter bezeichnet.

```
public interface IDtoView extends IsWidget {
   interface Presenter {
      void setDtoClicked(Long dtoId);
   }
   void setDtosToShow(Dto[] dtos);
}

public class DtoView extends Composite implements IDtoView {
   private static DtoViewUiBinder uiBinder = GWT.create(ConsoleViewUiBinder.class);

   @UiField
   ListBox dtoList;

   interface DtoViewUiBinder extends UiBinder<Widget, ConsoleView> {}

   public DtoView() {
      initWidget(uiBinder.createAndBindUi(this));
   }

   @Override
   public void setDtosToShow(Dto[] dtos) {
      //set list box with values
   }

   @UiHandler("dtoList")
   public void dtoSelected(ClickEvent e) {
      //parse element from listbox
      presenter.setDtoClicked(dto);
   }
   ...
}
```

Beispiel 6: Beispielimplementation einer GWT-View

Die ClientFactory ist nicht zwingend notwendig, jedoch beinhaltet sie statische Referenzen auf Views und dient als Speicher zur späteren Verlinkung zwischen Views. Ebenso in der ClientFactory gespeichert wird der EventBus, der zur Kommunikation der Mapper und Manager Klassen über verschiedene Activities hinweg dient und einen PlaceController um Wechsel zwischen Activities zu bewerkstelligen.

- Places, PlaceController und PlaceHistoryMapper [5]
  Places sind hauptsächlich Datenspeicher für Parameter die an Activities weitergegeben werden sollen. Places haben zusätzlich oft einen Tokenizer implementiert, der sich darum kümmert alle benötigten Parameter in eine URL umzuwandeln. Diese URL kann dann gespeichert werden und jederzeit im Browser wieder aufgerufen werden. Umgekehrt werden dann alle Parameter wieder in Objekte zurückgewandelt und stehen der Anwendung wieder zur Verfügung.
  Der PlaceHistoryMapper speichert sich diese generierten URL und kann diese dann auch wieder zur Verfügung stellen.
  Der PlaceController verbindet alle Places einer Anwendung mit den dazugehörigen Activities und ermöglicht einen Seitenwechsel in der Anwendung (z.B. mit Aufruf aus einer Activity: placeController.goto(new DtoPlace())).
- Activities, ActivityMapper und ActivityManager [5]
  Activities (Beispiel 7) enthalten nun die Clientseitige Businesslogik der Anwendung und implementieren das Presenter-Interface der View die sie bedienen. Sie werden gestartet und

müssen sich in der start-Methode bei der View registrieren (z.B. setPresenter(this)) und die View zum aktuellen Container (die gesamte Webseite oder einen Teilbereich der Seite) hinzufügen.

```
public class DtoActivity extends AbstractActivity implements IDtoView.Presenter {
   private IClientFactory clientFactory;
   private DtoPlace dtoPlace;
   private IDtoView dtoView;

   public DtoActivity(final IClientFactory clientFactory, final DtoPlace dtoPlace) {
      this.clientFactory = clientFactory;
      this.dtoPlace = dtoPlace;
      this.dtoView = clientFactory.getDtoView();
   }

   public void start(AcceptsOneWidget panel, EventBus eventBus) {
      dtoView.setDtosToShow(place.getDtos());
      panel.setWidget(dtoView);
   }

   @Override
   public void setDtoClicked(Long dtoId) {
      //do some business logic
   }
}
```

Beispiel 7: Beispielimplementation einer GWT-Activity

Der ActivityManager enthält den Überblick über alle laufenden Activities und reagiert auf Place-Wechsel. Er fragt bei den aktuell laufenden Activities nach, ob der Place-Wechsel überhaupt erlaubt ist und kann gegeben falls ein umschalten verhindern. Wenn ein Umschalten erlaubt ist, beendet er die aktuelle Activity und ruft mittels ActivityMapper und dem Place eine neue Activity auf. Beispiel 8 zeigt eine typische Implementierung eines solchen ActivityMappers.

```
public class WebInterpreterActivityMapper implements ActivityMapper{
   private final IClientFactory clientFactory;

   public WebInterpreterActivityMapper(IClientFactory clientFactory) {
      this.clientFactory = clientFactory;
   }

   public Activity getActivity(final Place place) {
      if (place instanceof DtoPlace) {
         return new DtoActivity(clientFactory, (DtoPlace) place);
      }
      //other possible places and activities
      return null;
   }
}
```

Beispiel 8: Beispielimplementierung einer GWT-ActivityMapper Klasse

Übersetzung von GWT-UI-Klassen in HTML Elemente

Das folgende Beispiel einer UI-Konfiguration zeigt die Verwendung von UI-Templates für GWT-UI-Komponenten. Zu jeder View oder UI-Komponente wird ein eine *.ui.xml-Datei abgelegt die die Formatierung und Gestaltung der Komponente festlegt wie Beispiel 9 zeigt.

```
<g:FlowPanel addStyleNames="panel">
   <g:Label ui:field="promptLabel" text=">"/>
</g:FlowPanel>
```

Beispiel 9: XML-Definition einer UI-Komponente in GWT

Ein FlowPanel und Label werden zwar auf einen -Tag abgebildet, sie können jedoch unterschieden werden durch unterschiedliche CSS-Klassen. Zusätzlich sind bei Label andere Attribute erlaubt, wie zum Beispiel das Einfügen von Text (text=“>“).

Mit dem Attribut „ui:field“ referenziert man auf eine Variable innerhalb der implementierenden View-Klasse. Mittels UiBinder werden diese dann verknüpft. Wenn sich nun die Variable ändert, wird automatisch der generierte HTML-Code ausgetauscht und die aktualisierten Werte angezeigt.

Zusätzlich zu diesen einfachen Elementen die von GWT zur Verfügung gestellt werden, können beliebige komplexe Komponenten eingebettet werden. Diese können wiederum aus einer einen ui.xml-Datei und dessen Konfiguration bestehen.

3. Programmstruktur

Die Webkonsole besteht aus den funktionalen Komponenten Plux, WebInterpreter-Plugin, WebSockets, WebSocketCommandInterpreter und den GWT-Klassen ConsoleView und –Activity, die auf Server und Client verteilt sind. Die JSON Komponente übernimmt Kodierung und den Austausch der Objekte zwischen Server und Client.

Abbildung 5 zeigt die Architektur der Webkonsole für einen WebClient. Verbinden sich mehrere WebClients entsteht eine 1:n-Verbindung, die es jedem WebClient erlaubt Befehle an den WebInterpreter zu senden und es genauso dem Server ermöglicht allen WebClients Benachrichtigungen des Plux-Systems zu übermitteln.

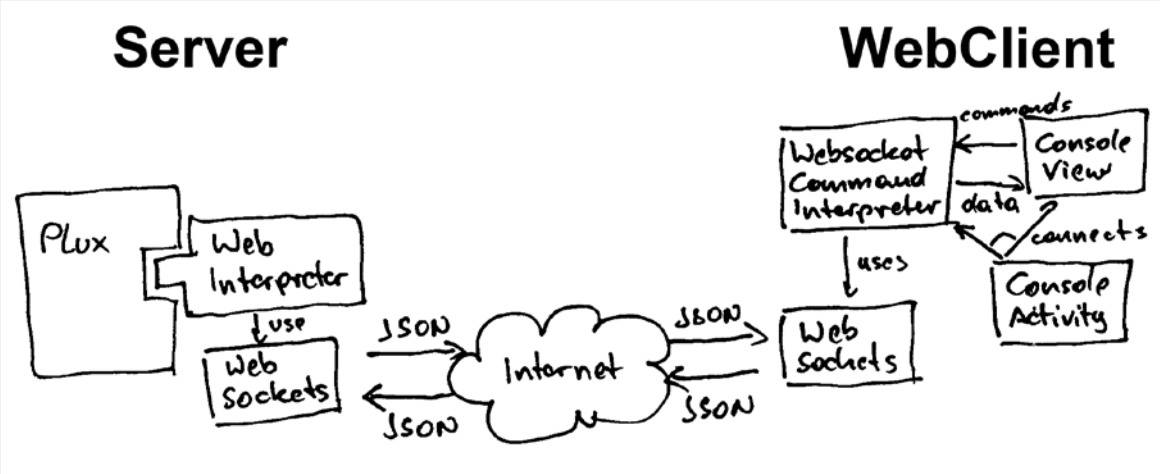

Abbildung 5: Systemübersicht

3.1. WebInterpreter (C#)

Serverseitig ist die Funktionalität als Plux-Plugin umgesetzt. Dieses Plugin (WebInterpreter) kann einfach in das Plugins-Verzeichnis der Plux-Instanz installiert und anschließend wie jedes andere Plux-Plugin verwendet werden. Die Übersicht der Komponenten und deren Verwendung ist in Abbildung 6 zu sehen.

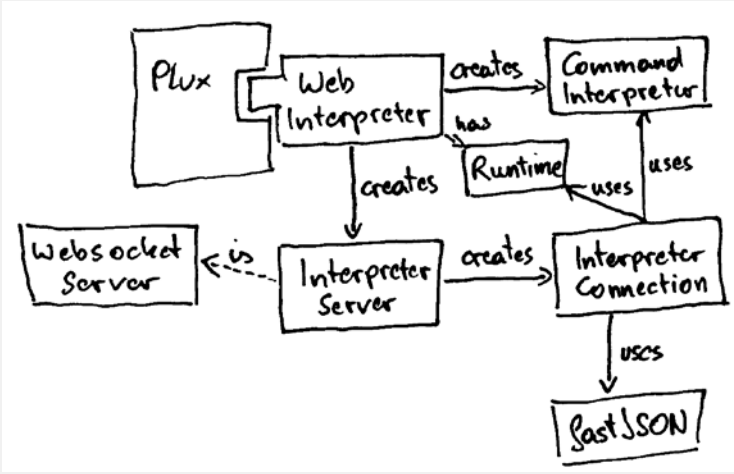

Abbildung 6: Detailübersicht WebInterpreter (C#) als Plux-Plugin

Das WebInterpreter-Plugin erstellt bei Start einen WebSocket-Server (InterpreterServer). Als Parameter für das Plugin werden die Konsolenkommandos eingeplugged.

Mittels Plux-Runtime wird für jede WebSocket-Verbindung (InterpreterConnection) ein CommandInterpreter gestartet, der die Befehle entgegen nimmt und in der Runtime ausführt.

Jede InterpreterConnection führt die empfangenen Kommandos aus. Gibt es einen Rückgabewert der ausgeführten Funktion, wird diese automatisch an den entsprechenden Client zurückgesendet, wie z.B. der Aufruf des complete-Commands. Erfolgt der Aufruf asynchron (z.B. Aufruf des execute-Commands) wird das Resultat an alle verbundenen InterpreterConnections als Broadcast ausgesendet.

Das Senden und Empfangen der Nachrichten funktioniert mittels asynchroner Methoden der WebSocket-Schnittstelle. Die empfangenen Daten entsprechen dem JSON-Format und werden Hilfe der fastJSON-Bibliothek in ein MessageItem-Objekt umgewandelt. Ebenso generiert fastJSON aus MessageItem-Objekten einen JSON-String, der dann wiederum über den WebSocket verschickt werden kann.

3.2.WebClient (Java-GWT)

Clientseitig soll die Bedienungsoberfläche möglichst jener Plux-Konsole entsprechen. Dem entsprechend einfach ist auch der folgende UI-Mockup in Abbildung 7 der Applikation:

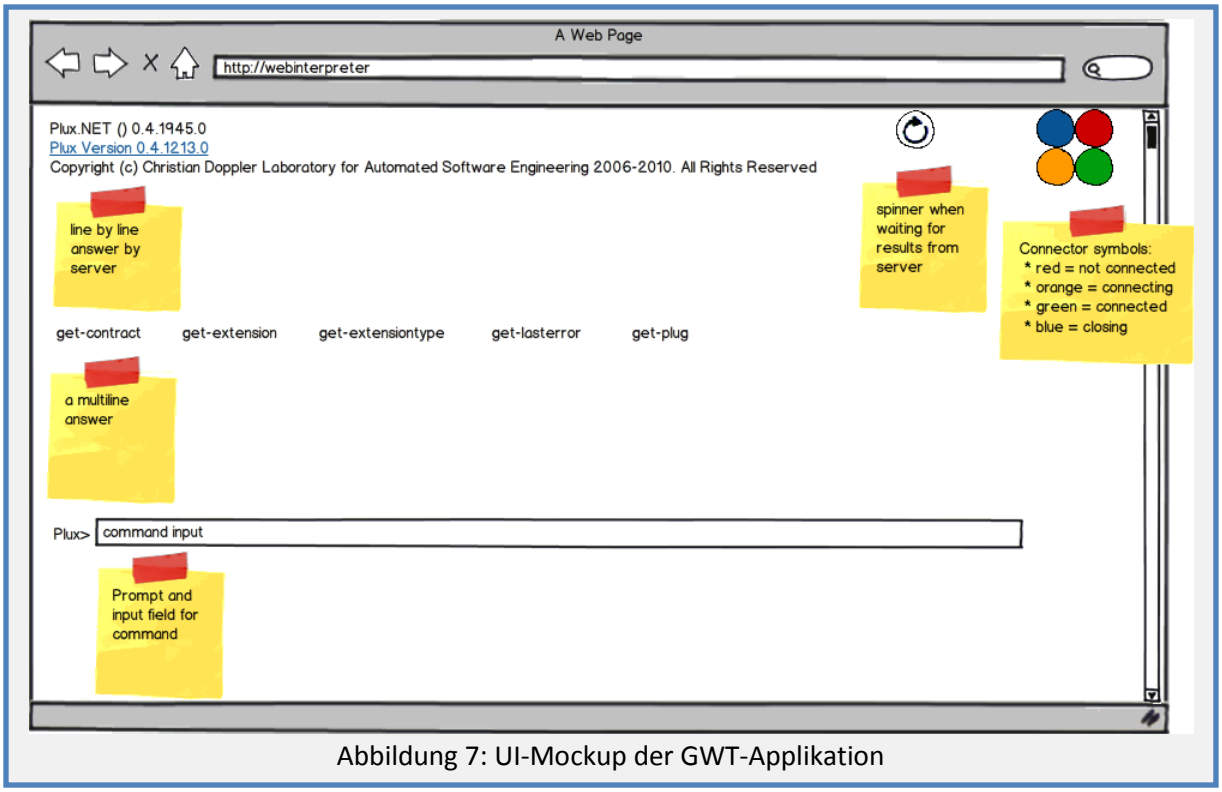

Abbildung 7: UI-Mockup der GWT-Applikation

Zusätzlich zu den gewohnten Elementen einer Konsole befinden sich im rechten oberen Eck noch Symbole für die Konnektivität der WebSocket-Verbindung. Einerseits wird dort angezeigt in welchem Status sich der WebSocket befindet, andererseits ob der Client gerade noch auf eine Antwort des Servers wartet.

Für die Entwicklung als GWT-MVP-Applikation dient PluxWebInterpreter als Einstiegspunkt, der die gesamte Web-Anwendung initialisiert. Der Place wird hier allerdings nur verwendet um der Architektur zu genügen. Ein URL-Mapping und Browser-History Unterstützung ist nicht sinnvoll umsetzbar, da jeder Befehl den Status des Servers ändert und sich dadurch nicht der Originalzustand wieder herstellen lässt. Abbildung 8 zeigt eine Detailübersicht über die Architektur der Client Komponente.

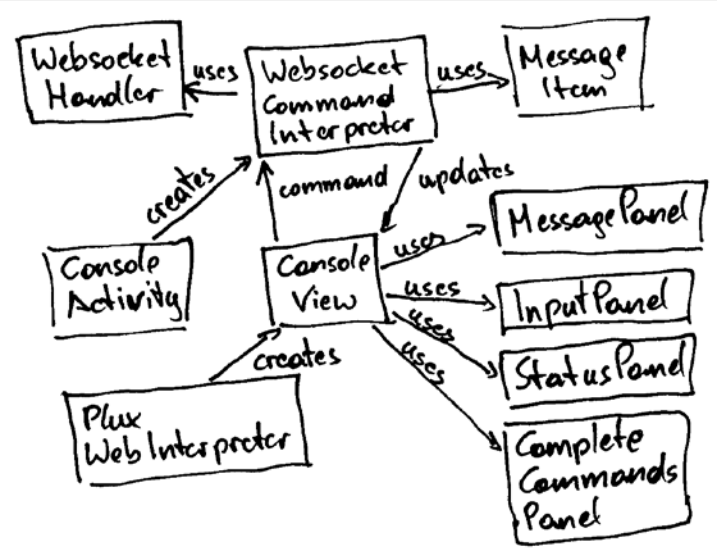

Abbildung 8: Detailübersicht des WebClients (Java) als GWT-Applikation

Die ConsoleView enthält alle nötigen Elemente um einzelne Zeilen des Interpreters anzuzeigen. Ebenso werden die Status des WebSockets angezeigt und ein Eingabefeld zur Verfügung gestellt.

Die ConsoleActivity enthält in unserem Fall keinerlei Businesslogik. Sie verknüpft nur den WebSocketCommandInterpreter mit der ConsoleView. Die weiteren Funktionalitäten sind somit vom GWT-Lifecycle-Objekt der Activity getrennt und austauschbar.

Der WebSocketCommandInterpreter nimmt die eingegebenen Befehle der ConsoleView entgegen und packt diese in ein MessageItem, welches anschließend über den WebSocket an den Plux-Server geschickt wird.

Zum Verschicken muss zuvor noch das MessageItem in ein JSON-Objekt umgewandelt werden. Dies lässt sich einfach mit dem von GWT mit gelieferten TransferBean bewerkstelligen.

4. Implementierung

4.1. WebInterpreter als Plux-Plugin

Das WebInterpreter-Plugin ist wie in Beispiel 9 definiert und ist dann wie in Abbildung 10 in das Plux-Framework integriert.

```
[Extension]
[Plug("Application")]
[Slot("Plux.CommandLibrary",
    OnPlugged = "AddCommandLibrary",
    OnUnplugged = "RemoveCommandLibrary")]
public class WebInterpreter : ExtensionBase, IApplication
{
...
}
```

Beispiel 10: Plux Plugin-Definition

Es erfüllt den Plug „Application“ um gestartet zu werden. Als Slot wird die CommandLibrary eingefügt, die alle nötigen Kommandos der Plux-Konsole beinhaltet und so auch für CommandInterpreter dieses Plugins zur Verfügung steht.

Als Basisklasse wird ExtensionBase verwendet, um Zugriff auf die Plux-Runtime zu erhalten in der die Kommandos ausgeführt werden. Ebenso ermöglicht sie einen Zugriff auf die Stop-Funktion des Plugins um den WebSocket-Server entsprechend zu stoppen.

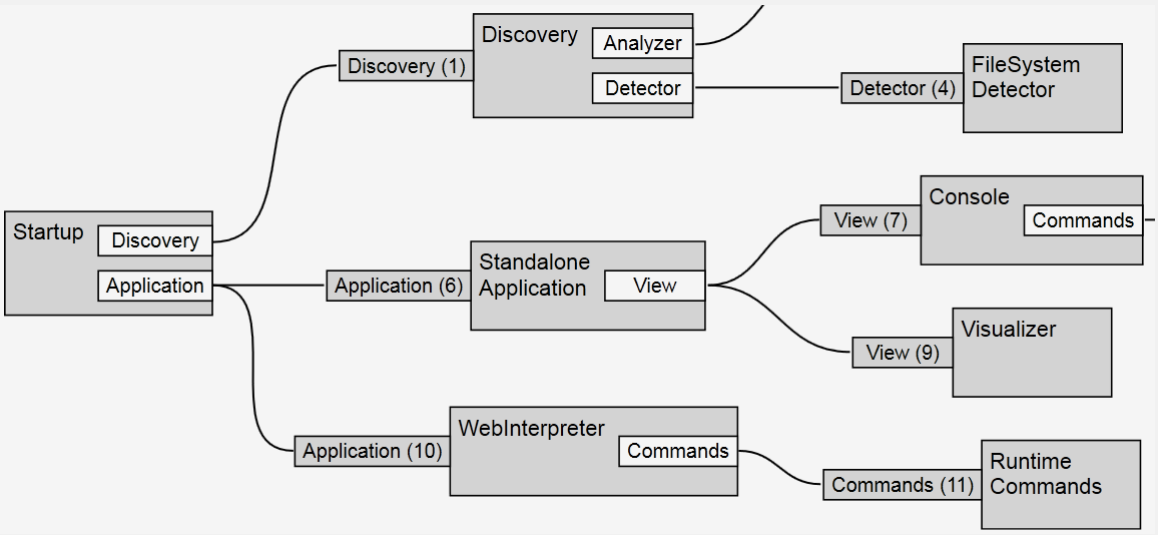

Abbildung 9: Installiertes WebInterpreter Plux-Plugin

4.2.JSON-Serialisierung

JSON [12] dient als nützliches Formatierungsprotokoll des zu übertragenden Objektes.

Serverseitige JSON-Behandlung

Die JSON-Serialisierung erfolgt serverseitig mit der Bibliothek fastJSON [13] die es über Reflection ermöglicht Objekte ohne weitere Spezifikation zu de-/serialisieren. Die verwendete Version v1 hatte jedoch noch ein paar Probleme Objekte korrekt umzuwandeln. Ursprünglich wurden bei der Deserialisierung keine Enumerationen und generischen List-Interfaces unterstützt. Diese generischen Listen können wie in Beispiel 10 dynamisch erstellt werden..

```
if (pt.IsInterface) {
   Type listGeneric =
pt.GetGenericArguments().Length>0?pt.GetGenericArguments()[0]:typeof(object);
   Type openType = typeof(List<>);
   Type closedType = openType.MakeGenericType(listGeneric);
   col = (IList) Activator.CreateInstance(closedType);
}
```

Beispiel 11: Erstellen einer generischen Liste mittels Reflection

Es wird zuerst der Generische Typ extrahiert, eine Liste ohne definierten Typ erstellt, diese daraufhin typisiert und mit Activator.CreateInstance ein Objekt erzeugt.

Weiters wurde vorausgesetzt, dass Listen immer einen komplexen Typ besitzen. So wurden simple Datentypen wie String nicht korrekt umgewandelt. Für diese Arbeit wurde die Abfrage erweitert (Beispiel 11) ob es sich um einen simplen Typen handelt, der dann mit der Funktion ChangeType korrekt umgewandelt werden kann. Ansonsten wird das Objekt rekursiv aufgelöst.

```
if (!ob.GetType().FullName.Contains("Dictionary"))
   col.Add(ChangeType(ob,ob.GetType()));
else
   col.Add(ParseDictionary((Dictionary<string, object>)ob));
```
Beispiel 12: Abfrage eines simplen Datentyps

Die Enum-Erweiterung betrifft nun genau die bereits oben genannte Methode ChangeType() und ist im nächsten Beispiel 12 dargestellt.

```
if (conversionType.IsEnum)
   return Enum.Parse(conversionType, (string)value);
else
   return Convert.ChangeType(value, conversionType);
```
Beispiel 13: Erstellen einer generischen Liste mittels Reflection

System.Convert.ChangeType() erlaubt die Konvertierung von simplen Objekten, jedoch nicht Enums. Diese müssen mit der Klasse System.Enum.Parse() passieren. Diese nimmt die Stringrepräsentation des Objektes und versucht den entsprechenden Enum des angegeben Typs zu erstellen.

Zur Serialisierung des Objektes war die Methode WriteObject() nicht korrekt umgesetzt. Die einzelnen Variablen wurden nicht Komma seperiert eingefügt und nur mit einem Leerzeichen verkettet.

Das Handling ist dann mit den Funktionen ToJson() und ToObject() sehr einfach zu bewerkstelligen (Beispiel 13).

```
jsonString = fastJSON.JSON.Instance.ToJSON(message)
message = (MessageItem) fastJSON.JSON.Instance.ToObject(jsonString);
```
Beispiel 14: Verwendung der fastJSON-Bibliothek

Clientseitige JSON-Behandlung

Da es sich bei JSON wie der Name bereits verrät um einen JavaScript Object Notation handelt, bietet GWT umfangreiche Möglichkeiten diese Art von Objekten zu nutzen. Zur Deserialisierung wird der von GWT mitgelieferte JSON-Parser benutzt (Beispiel 14).

```
JSONObject jsonObject = JSONParser.parseLenient(jsonString).isObject();
messageItemType = jsonObject.get("$type").isString().stringValue();
obj.setMessageList(jsonObject.get("Messages").isArray());
obj.setMessageType(MessageType.valueOf(jsonObject.get("MessageType").isString().stringValue
()));
obj.setConsoleType(ConsoleType.valueOf(jsonObject.get("ConsoleType").isString().stringValue
()));
```
Beispiel 15: Parsen eines JSONString in GWT

Das Attribut $type wird von fastJSON mitgeliefert und beinhaltet die Assembly-Information des MessageItem-Objektes in C#. Dies wird dann wieder genutzt die entsprechenden Objekte zu erzeugen und wird deshalb in GWT mit übernommen, jedoch nirgends verwendet.

JSON-Strings werden mittels typisierten AutoBeanFactory erzeugt. Dazu ist nur ein typisiertes Interface nötig und wird später im Code mittels GWT.create(TransferAutoBeanFactory.class) instanziiert (Beispiel 15).

```
public interface TransferAutoBeanFactory extends AutoBeanFactory{
   AutoBean<IMessageItem> messageItem();
   AutoBean<IMessageItem> messageItem(IMessageItem toWrap);
}
private String generateJSON(MessageItem mi) {
   mi.set$type(messageItemType);
   TransferAutoBeanFactory factory = GWT.create(TransferAutoBeanFactory.class);
   AutoBean<IMessageItem> bean = factory.messageItem(mi);
   String jsonString = AutoBeanCodex.encode(bean).getPayload();
   return jsonString.toString()
                    .replace("messageType", "MessageType")
                    .replace("consoleType", "ConsoleType")
                    .replace("messages", "Messages");
}
```

Beispiel 16: Serialisieren eines Objektes in GWT

Die Klasse AutoBeanCodex.encode() generiert nun aus dem AutoBean den gewünschten JSON-String. Bevor dieser jedoch versendet werden kann, müssen die Variablennamen noch etwas geändert werden. Durch die unterschiedliche Konvention in C#, in der auch Variablen und Methoden mit einem Großbuchstaben beginnen, müssen diese nun im JSON-String geändert werden, da diese später bei der Deserialisierung in C# nicht erkannt werden würden.

Das Interface IMessageItem wird hier nur gebraucht, um das TransferAutoBeanFactory zu instanziieren, da für GWT-Anwendungen Interface und Implementierungen möglichst stark getrennt werden um mit weiterführenden Methoden wie GIN Dependency Injection zu betreiben. In diesem Fall enthält es jedoch nur Getter und Setter die das MessageItem vollständig beschreiben.

4.3.InterpreterServer als WebSocket-Server

WebSockets sind zurzeit noch keinem Standard unterlegen. Es existieren bereits viele Drafts und ein Standard ist absehbar. Browser unterstützen dabei unterschiedliche Drafts, wenn sie es überhaupt tun. Dieser Zustand erschwert es eine geeignete WebSocket-Server-Bibliothek zu finden.

Zuerst wurde ein älterer OpenSource WebSocket-Server [11] angepasst, sodass er dem Hixie-Draft-76 (Hybie-00) entsprach. Dieser Draft wurde allerdings nur von Chrome (bis Version 9) unterstützt. Mit der Umstellung auf Chrome v10 und FireFox 7 wurde in beiden Browsern die Unterstützung der WebSockets auf Protokollversion 8 angehoben. Somit war der alte Server nicht mehr kompatibel mit den neuen Browsern.

Eine Anpassung war zwar möglich, da das OpenSource Projekt jedoch nicht mehr aktiv weiterentwickelt wurde, müsste man jede Protokolländerung mitziehen.

Im Verlaufe des Projektes entstand neues OpenSource Projekt von Bauglir [10], der ohne Änderungen die Protokollversion 8 unterstützt und aktiv weiterentwickelt wird.

Nach Start des Servers wird für jede eingehende Verbindung eine InterpreterConnection erstellt, die eine Referenz auf den CommandInterpreter und die Plux-Runtime beinhaltet. Die InterpreterConnection ist im Weiteren für die gesamte Kommunikation mit einem Web-Client verantwortlich.

Wenn die InterpreterConnection gestartet wird, wird zuerst die VersionInfo und ein Prompt an den Client gesendet (siehe Abbildung 10) . Mittels der asynchronen Methode ConnectionRead wartet nun die InterpreterConnection auf Nachrichten des Clients. Erreicht ihn eine Nachricht wird diese zuerst deserialisiert (siehe 4.2) und dann auswertet.

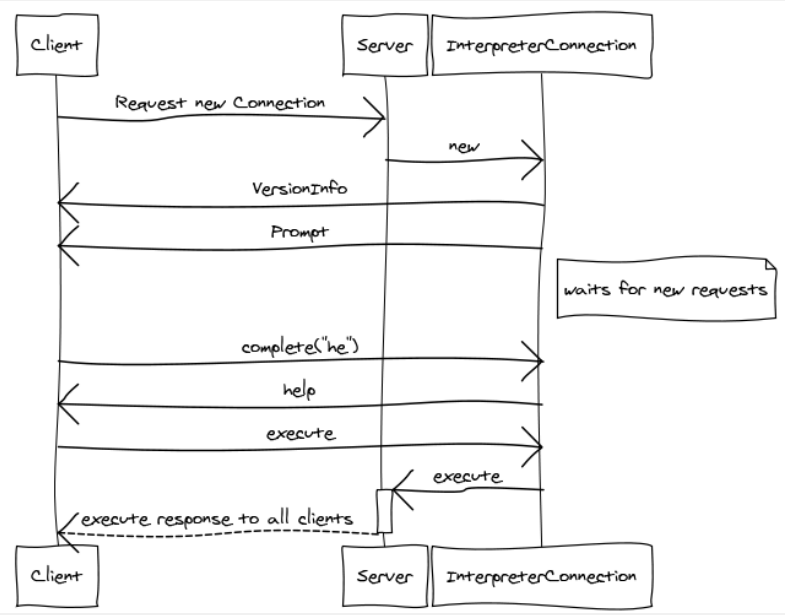

Abbildung 10: Sequenzdiagramm [21] der Kommunikation zwischen Client, Server und InterpreterConnection

Handelt es sich um ein Complete-Command wird dieses sofort im CommandInterpreter ausgeführt und der Rückgabewert, sofern dieser existiert, direkt an den Client zurückgeschickt (Beispiel 16).

```
string cc = interpreter.Complete(message.Messages[0], message.SearchAnywhere);

if (cc!= null && cc.Length > 0) {
   SendMessage(cc, MessageType.Normal, ConsoleType.Complete);
}
```

Beispiel 17: Abhandlung eines Compelete-Commands

Bei Ausführen eines Kommandos wird dieses in der Runtime mittels Dispatcher aktiviert (Beispiel 17).

```
private delegate void InterpreterExecute(String param);

private void ExecuteCommand(MessageItem message) {
   runtime.Dispatcher.Invoke(new InterpreterExecute(interpreter.Execute),
                             new object[] { message.Messages[0] });
}
```

Beispiel 18: Abhandlung eines Execute-Commands

Zum Ausschicken des Ergebnisses, wird die Klasse WebConsoleWriter verwendet. Dieser implementiert IConsoleWriter und wird dem Interpreter hinzugefügt. Er wird automatisch aufgerufen, wenn sich im Buffer der Konsole etwas befindet.

Da nicht mehr Rückgeschlossen werden kann, von welchem Client der dazugehörende Aufruf kam, wird wir ein Broadcast an alle verbundenen Clients durchgeführt (Beispiel 19).

```
String jsonString = fastJSON.JSON.Instance.ToJSON(
                                new MessageItem(items, type, nextCommand));
if (!fServer.IsRunning) return;
fServer.LockConnections();
for (int i = 0; i < fServer.ConnectionCount; i++)
{
   fServer.GetConnection(i).SendText(jsonString);
}
fServer.UnlockConnections();
```

Beispiel 19: Broadcast an alle Clients

4.4.WebSocket Handler als Java-JavaScript

GWT bietet mittels native-JavaScript Methoden direkten Zugriff auf alle JavaScript Funktionalitäten des Browser. Beispiel 19 zeigt einen solchen Java-Wrapper für native JavaScript Funktionalität.

```
public final native void send(String data) /*-{
   this.send(data);
}-*/;
```

Beispiel 20: Native JavaScript Methode in GWT

Der WebSocket für GWT implementiert API der W3C (siehe 2.3). Die Funktionalität ist in einem eigenen Modul gekapselt und aus der OpenSource-Bibliothek gwt-comet [3] adaptiert. Eine Änderung war nötig, da Firefox keine WebSocket-Klasse unterstützen, da diese noch kein fertiger Standard ist.

Die Änderung beschränkt sich auf den Konstruktor der Java-WebSocket Klasse in der der UserAgent des Browsers ausgelesen wird und das WebSocket-Objekt entsprechend erzeugt wird (Beispiel 20).

```
public static native WebSocket create(String url) /*-{
   if (navigator.userAgent.match(/(Firefox)/) == null) {
      return new WebSocket(url);
   } else {
      return new MozWebSocket(url);
   }
}-*/;
```

Beispiel 21: WebSocket Konstruktor für FireFox und Chrome

Zur leichteren Verwendung wurde noch die Methode createWebSocketIfSupported() hinzugefügt. Sollte ein Browser keine WebSockets unterstützen, wirft dieser eine Exception. Hier wird diese gefangen und ein null-Wert zurück geliefert. Dies ist entsprechend, wie weitere HTML5-Funktionalitäten in GWT (z.B. clientseitiger Speicher) umgesetzt.

Mit den bereitgestellten Listener-Interfaces für onError, onMessage, onOpen und onClose lassen sich dann bequem über Java-Methoden die Ereignisse abfangen und bearbeiten (siehe Beispiel 21).

```
private class WebSocketHandler implements ErrorHandler, OpenHandler,
                                          CloseHandler, MessageHandler {
   @Override
   public void onClose(IWebSocket webSocket) {
      //onClose handling
   }

   @Override
   public void onError(IWebSocket webSocket) {
      //onError handling
   }

   @Override
   public void onOpen(IWebSocket webSocket) {
      //onOpen handling
   }

   @Override
   public void onMessage(IWebSocket webSocket, MessageEvent messageEvent) {
      //onMessage handling
   }
}
```

Beispiel 22: WebSocket Handler

4.5.WebClient als Java-GWT

Einstiegspunkt ist die Klasse PluxWebInterpreter, die wie bereits beschrieben (siehe 3.2) alle nötigen Komponenten für eine GWT-MVP-Anwendung zusammenbaut. Die Kommunikation zwischen Client und Server wurde bereits in den vorangegangenen Kapiteln (4.2 und 4.4) beschrieben.

Anzumerken ist noch, das PluxWebInterpreter den Punkt festlegt (Beispiel 22), wo die Anwendung eingefügt werden soll, und mit welchen Parametern sich der WebSocket zum Server verbindet.

```
private final String defaultWsConnectionUrl = "ws://127.0.0.1:8181/pluxconsole";

public void onModuleLoad() {
   ...
   RootLayoutPanel.get().add(widget);
   String connectUrl = Window.Location.getParameter("connectUrl");
   ResourceFactory.createExecutionCommand(
           new ConnectCommandExecutor(connectUrl==null?defaultWsConnectionUrl:connectUrl));
   ...
}
```

Beispiel 23: Initialisieren der Anwendung

Mit RootLayoutPanel.get() wird der gesamte Bereich der Seite geladen, in der das Anwendungs-JavaScript geladen wurde und die Web-Konsole hinzugefügt. Durch genaueres Spezifizieren in der get-Methode durch einen id-String eines HTML-Elements, kann die Web-Konsole auch nur in einem Teilbereich der Seite geladen werden.

Mit Window.Location.getParameter() wird die connectUrl ausgelesen und verwendet, sollte diese gesetzt sein. Ansonsten wird zur default WebSocket-URL verbunden.

Der WebSocketCommandInterpreter dient nun als Businesslogik für den Client. Er gibt Befehle die in der Web-Konsole eingegeben wurden weiter an den WebSocket, und übergibt an die Web-Konsole Daten die angezeigt werden sollen. Des Weiteren besitzt der WebSocketCommandInterpreter ein eigenes Kommando „—connect“, welches bei Abbruch der WebSocket-Verbindung (z.B. durch Abbruch der Internetverbindung,…) einen neuen WebSocket anlegt und sich erneut zum Server verbindet.

Web-Konsole

Die Benutzeroberfläche der Web-Konsole besteht, wie in Abbildung 7 zu sehen aus mehreren Unterelementen. Diese sind das InputPanel, das StatusPanel, das CompleteCommandsPanel und das MessagePanel. Die Web-Konsole kümmert sich dabei um die richtige Zusammenstellung dieser Einzelelemente.

Das InputPanel (Beispiel 23) besteht aus einem Prompt und einer in der Größe veränderbaren HTML-TextArea. Diese wurde aus der gwt-traction [4] Bibliothek übernommen.

```
<g:FlowPanel addStyleNames="consoleInputPanel">
   <g:Label ui:field="promptLabel" text=">"/>
   <t:AutoSizingTextArea ui:field="consoleInputTextArea" maxFromCss="true" />
</g:FlowPanel>
```

Beispiel 24: InputPanel Ui-Code

Das StatusPanel (Beispiel 24) dient zur Anzeige des Verbindungsstatus und ob im Moment eine Anfrage an den Server gesendet wurde.

Ein DeckPanel wird wiederrum auf einen -Tag abgebildet. Der Unterschied liegt jedoch darin, dass für jedes DeckPanel jeweils nur ein Unterelement gleichzeitig angezeigt wird. Dies ist in unserem Fall besonders günstig, da sich z.B. der WebSocket sich nie in zwei verschiedenen Stati befinden kann.

```
<g:FlowPanel addStyleNames="statusPanel">
   <g:DeckPanel ui:field="deckPanelLoading" addStyleNames="status">
      <g:FlowPanel/>
      <g:Image ui:field="imageLoading" title="{msg.loading}"/>
   </g:DeckPanel>
   <g:DeckPanel ui:field="deckPanelWebsocketConnection" addStyleNames="status">
      <g:Image ui:field="imageConnecting" title="{msg.status_connecting}"/>
      <g:Image ui:field="imageConnected" title="{msg.status_connected}"/>
      <g:Image ui:field="imageClosing" title="{msg.status_closing}"/>
      <g:Image ui:field="imageClosed" title="{msg.status_closed}"/>
   </g:DeckPanel>
</g:FlowPanel>
```

Beispiel 25: StatusPanel Ui-Code

Das CompleteCommandsPanel zeigt eine Liste von möglichen Kommandooptionen an. Diese sollen in einer Tabellenartigen Struktur angezeigt werden (siehe Beispiel 25).

```
<g:FlowPanel ui:field="completeCommandsPanel" addStyleNames="completeCommandsPanel"/>
```

Beispiel 26: CompleteCommandsPanel Ui-Code

Um dieses Element möglichst flexibel zu halten, wird in der ui.xml-Datei nur ein FlowPanel referenziert, welches dann dynamisch mit neu erstellten Labels befüllt wird (siehe Beispiel 26).

```
for (String c:commands) {
   Label l = new Label(fillWithSpaces(c,maxChar));
   completeCommandsPanel.add(l);
}
```

Beispiel 27: Befüllen des CompleteCommandsPanel

Das Auffüllen mit Leerzeichen dient nur zum Ausgleich unterschiedlich langer Kommandos.

Im MessagePanel, das ebenfalls eine Liste von Nachrichten anzeigen kann, wird genauso verfahren wie im CompleteCommandsPanel.

In der Web-Applikation wird das Aussehen nicht mehr nur von der Struktur des HTML-DOM-Baums beeinflusst, sondern vielmehr durch Cascading Stylesheets (CSS). Diese besitzen jedoch eine Menge an Einschränkungen, die durch neuere Methoden erweitert wurden.

In diesem Projekt wird SASS (Syntactically Awsome Stylesheets) [1] verwendet, welches eine Erweiterung von CSS3 darstellt. Der Vorteil besteht hier darin, dass Variablen, Imports und vieles mehr verwendet werden kann. Dadurch wird es möglich Styling-Bibliotheken zu erstellen und diese dann für alle Elemente gleich anzuwenden (siehe Beispiel 27). Weiters wird SASS anschließend in eine CSS-Datei umgewandelt, die von jedem modernen Browser ohne Probleme verarbeitet werden kann.

```
_metrics.scss:
$lineSpacing: 0px;
$errorFontColor: red;

messagepanel.scss:
@import "../common/metrics";

.messagePanel-Error {
    font-weight: bold;
    color: $errorFontColor;
    margin-bottom: $lineSpacing;
}
```

Beispiel 28: SASS Definitionen

Die Übersetzung der SASS-Dateien in CSS-Dateien kann mittels JRuby und Ant-Skript erfolgen, welches auch bei einem automatischen Build des Systems im init-Vorgang definiert sein kann (siehe Beispiel 28).

```
<path id="ant.classpath">
   <pathelement path="${basedir}/lib/jruby-complete.jar"/>
</path>

<macrodef name="sass">
   <attribute name="from"/>
   <attribute name="to"/>
   <attribute name="style" default="compressed"/>
   <sequential>
      <java classname="org.jruby.Main" classpathref="ant.classpath" fork="true">
         <env key="GEM_PATH" path="${basedir}/lib/jruby"/>
         <arg line="-S ${basedir}/lib/jruby/bin/sass --style @{style} @{from} @{to}"/>
      </java>
   </sequential>
</macrodef>

<target name="init" description="Build initialization">
   <sass from="style.scss" to="style.css"/>
</target>
```

Beispiel 29: SASS Kompilierung mit JRuby und Ant

Im Zusammenspiel aller Komponenten ergibt sich für die Web-Konsole Abbildung 11.

Abbildung 11: Laufende Web-Konsole

5. Beurteilung

Die große Herausforderung dieses Projektes bestand darin, dass viele verschiedene Technologien verwendet werden mussten, um eine Web-Konsole für das Plux.NET-Framework zu schaffen.

Besonders muss das Augenmerk auf die Kommunikation zwischen Server und Client gelegt werden. Hier wurde mit WebSockets eine Technologie verwendet, die erst in den Kinderschuhen steckt und noch ständig bis zu einem Standard weiterentwickelt wird. Sie hat jedoch das Potenzial Web-Anwendungen durch eine bidirektionale Verbindung zwischen Server und Client zum Datenaustausch zu revolutionieren. Web-Anwendungen können durch diese neuen Kommunikationsmöglichkeiten verstärkt wie Desktop-Applikation wirken und diese in manchen Bereichen verdrängen.

Die Entwicklung eines GWT-Clients für diese Problemstellung ist teilweise nicht gerechtfertigt, da mit dem GWT-Framework ein Overhead mit übernommen werden muss, welcher bei einer reinen problemspezifischen JavaScript-Anwendung wegfallen würde.
Nichts desto trotz ermöglicht es GWT die komfortable Programmierung von Web-Anwendungen in Java ohne sich je mit JavaScript und dessen Fallen beschäftigen zu müssen. Mit wachsender Größe der Anwendung wird der Einsatz von GWT immer gerechtfertigter, da es auch eine breite Integration mit anderen Java-Technologien wie Spring auf Serverseite bietet. Zudem lassen sich verstärkt Programmiermuster aus der Objektorientierung einsetzen, die eine Wiederverwendung von Code verbessern.

Entwickelt wurde der C#-Teil als Plux.NET-Plugin mit Microsoft Visual C# 2010 Express [15]. Diese freie Version ermöglicht einen guten Einstieg in die C#-Programmierwelt. Die Express-Edition schränkt den Funktionsumfang des Microsoft Visual Studio 2010 teilweise stark ein. Dies wirkt sich bei dieser Arbeit besonders bei dem Starten des erstellten Programmes aus. Das Plugin wird als DLL-Datei vom Compiler gebaut. Dieses kann dann nicht automatisch in den Pfad der Plux-Anwendung kopiert. Zusätzlich kann kein ausführbares Programm definiert werden, welches automatisch gestartet werden soll nach erfolgreichen übersetzen des Plugins.

Die GWT-Entwicklung wurde mit der Entwicklungsumgebung IntelliJ Idea 10.5 Ultimate Edition [16] eingesetzt. Die ebenso erhältliche Free Community Edition unterstützt keine Webentwicklungs-Frameworks und ist daher ungeeignet [17]. Alternativ kann die SpringSource Tool Suite [18] basierend auf Eclipse genutzt werden.
IntelliJ Idea bietet jedoch eine bessere Unterstützung in Form von Code-Completion und Analyse von GWT-Syntax (Referenzen zwischen ui.xml-Dateien und den entsprechenden Implementierungsklassen, die in Eclipse nicht vorhanden ist).

A. Literaturverzeichnis

GWT & Web:
[1] http://sass-lang.com/
[2] http://code.google.com/webtoolkit/
[3] http://code.google.com/p/gwt-comet/
[4] http://code.google.com/p/gwt-traction/
[5] http://code.google.com/webtoolkit/doc/latest/DevGuideMvpActivitiesAndPlaces.html
WebSockets:
[6] http://en.wikipedia.org/wiki/WebSocket
[7] http://tools.ietf.org/html/draft-ietf-hybi-thewebsocketprotocol-10
[8] http://www.tavendo.de/autobahn/testsuite/report/clients/index.html
[9] http://dev.w3.org/html5/websockets/
C#-WebSocket-Server:
[10] http://code.google.com/p/bauglir-websocket/
[11] http://www.codeproject.com/KB/webservices/c_sharp_web_socket_server.aspx
JSON
[12] http://www.json.org/
[13] http://www.codeproject.com/KB/IP/fastJSON.aspx
Pux.NET
[14] http://ase.jku.at/plux/
Programmierumgebungen
[15] http://www.microsoft.com/visualstudio/en-us/products/2010-editions/visual-csharp-express
[16] http://www.jetbrains.com/idea/
[17] http://www.jetbrains.com/idea/features/editions_comparison_matrix.html
[18] http://www.springsource.com/developer/sts
Sonstige
[19] http://ase.jku.at/
[20] http://ssw.jku.at/
[21] http://www.websequencediagrams.com/

B. Abbildungsverzeichnis

C. Beispielverzeichnis