Bachelorarbeit

Andreas Winschu

Entwicklung einer Editor-Webapplikation zur Pflege eines ontologiebasierten Patientendialogsystems

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Andreas Winschu

Entwicklung einer Editor-Webapplikation zur Pflege eines ontologiebasierten Patientendialogsystems

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer : Prof. Dr. rer. nat. Michael Neitzke
Zweitgutachter : Prof. Dr. Stefan Sarstedt
Abgegeben am 07. Juli 2010

Andreas Winschu

Thema der Bachelorarbeit

Entwicklung einer Editor-Webapplikation zu Pflege eines ontologiebasierten Patientendialogsytems

Stichworte

Rich Internet Application, Ontology, Intelligente Systeme, Flex, ActionScript, MXML, Presentation Model, Dependency Injection, Parsley

Kurzzusammenfassung

Im Rahmen einer Zusammenarbeit von Alexander Bokov und der Techniker Krankenkasse ist ein Informationsystem in Form eines Webdialogs entstanden. Das System entscheidet aus den Benutzerneigungen über den Verlauf des Dialogs. Hierfür pflegt es eine Ontologie Wissensbasis. Nun ist eine Authoring-Software notwendig, mit der Themenexperten den Dialog mit Inhalten füllen können. Die Authoring-Software muss ebenfalls eine Webanwendung werden, jedoch gleichzeitig desktopähnliches Verhalten mitbringen. Diese Arbeit beschäftigt sich mit der Erstellung einer solchen Software.

Andreas Winschu

Title of the paper

Development of an editor-webapplication for administration of a dialoguesystem for patients.

Keywords

Rich Internet Application, Ontology, Intelligent Systems, Flex, ActionScript, MXML, Presentation Model, Dependency Injection, Parsley

Abstract

During a cooperation between Alexander Bokov and Techniker Krankenkasse an Informationsystem in form of a webdialogue was developed. The systems makes the Decisions about the progress of the dialogue from the users preferences. In order to achieve this, it administrates an ontology knowledge-base. Now an athoring-software is required, which provides the ability for an expert of a special topic to fill the dialogue with contents. The authoring-software must also be an webapplication, however it should provide desktop-smilar functionality as well. This thesis concerns itself with the development of such a software.

Inhaltsverzeichnis

1. Einleitung

In der Informatik ist es üblich bestimmte Entwicklungsstadien eines Software Produkts mit Versionsnummern zu versehen. Und obwohl das Web kein wirkliches Software Produkt ist und seine Kommunikationsprotokolle sich seit seiner Entstehung kaum verändert haben, so hat die Art und Weise wie das Web als Medium genutzt wird dazu geführt das in Expertenkreisen das Web mittlerweile mit der Versionsnummer 4 oder gar höher beziffert wird. Dabei wird sich Jeder einig sein, dass der Wandel von Auslieferung vom statischen Inhalt zu dynamisch, durch Benutzer veränderbaren Webseiten die Versionsnummer 2 eingeleitet hat. Die weiteren Versionierungen sind dagegen weniger eindeutig.

Ein Stichwort in dieser Entwicklungslinie des Internets ist das „Semantic Web". Darunter versteht man die Disziplin Informationen im Internet in maschinenlesbarer Form bereitzustellen. Mit Hilfe von Methoden aus der künstlichen Intelligenz können die Maschinen die Bedeutung dieser Information kognitiv verknüpfen.

Ebenfalls eine Versionsnummer könnte sich die Vorstellung, Software wie ein Service über das Internet auf Anfrage ausliefern zu können, sichern . Idealerweise sollte die Webanwendung dabei das „Look and Feel" einer herkömlichen Desktopanwendung beibehalten. Eine solche Webanwendung wird auch oft „Rich Internet Appliaktion", kurz RIA genannt.

Der Inhalt dieser Arbeit wird sich primär mit Erstellung einer RIA befassen, aber auch unvermeidbar durch die Motivation der Arbeit Themengebiete des „Semantic Web" betreffen.

1.1 Motivation

Man stelle sich vor ein Informationssystem wäre in der Lage, Schlussfolgerungen über die Neigungen der Benutzers zu ziehen und ihm passende Informationen zu liefern. Aus genau dieser Vorstellung ist im Rahmen einer Abschlussarbeit von Alexander Bokov in Zusammenarbeit mit Techniker Krankenkasse ein Webdialogsystem entstanden. Die Motivation dahinter bestand den Patienten eine Wissenbasis über verschiedene Erkrankungen und Therapien online bereitzustellen. Das besondere an diesem Dialogsystems ist, dass die Entscheidung, welche Fragen es dem Benutzer überhaupt stellt und welche Informationen es ihm anschließend liefert, erst im Verlauf des Dialogs getroffen wird. Um diese künstliche Intelligenz zu erzeugen, bedient sich Alexander Bokov der Wissenrespräsentation in Form von Ontolgien.

Selbstverständlich ist das System kein Ersatz für einen lebenden Experten und damit es in der

Lage ist, seine Schlussfolgerungen zu bilden, muss seine Datenbasis vom Entwickler gefüllt werden. Der Entwickler definiert welche Inhalte es gibt, welche Fragen gestellt werden können und welche Beziehungen als Resultat auf die Dialogantworten in die Wissenbasis wandern. Die Problematik dabei ist, dass ein Entwickler, der dieses System konfigurieren könnte, leider nur auf einem Gebiet Experte ist, und zwar im Entwickeln von Software. Und selbst für so einen Menschen wäre diese Datenbasis nach einiger Zeit kaum zu überschauen. Es wird also eine Administrationswerkzeug benötigt, welches auch ohne Kenntniss des „Semantic Web" Technologien bedient werden kann.

Eine gute Webanwendung bringt stets ihr eigenes Administrator-Werkzeug mit. Somit werden komplexe Installationen, Updatemechanismen oder Serverkonfigurationen erspart. In diesem Fall reichen aber die Möglichkeiten simpler HTML Formulare nicht aus, um die komplexen Datengebilde anschaulich visuell einem technisch nicht affinem Benutzer zu präsentieren. Benötigt wird volle Desktopfunktionalität in einer Webanwendung und, wie man aus der oberen Einleitung weiß, eine RIA.

1.2. Aufgabestellung

Das Ziel dieser Arbeit ist es eine Editor Applikation zu erstellen, welche es einem Autor erlaubt das von Alexander Bokov entwickelte „Semantic Web" Dialogsystem mit Inhalten zu befüllen. Dabei soll diese Editor Applikation ebenfalls eine Webanwendung sein und mit Dialogsystem mitausgeliefert werden.

1.3. Aufbau

Kapitel 2 „Grundlagen" legt die erforderlichen Grundlagen für diese Arbeit fest.
Kapitel 3 „Analyse" analysiert das Dialogsystem von Alexander Bokov und leitet die Anforderungen an die Editorappliktaion her
Kapitel 4 „Design" legt unter Berücksichtigung der gefundenen Anforderungen den Entwurf für eine graphischen Prototyp-Implementierung fest.
Kapitel 5 „Architektur" liefert ein Konzept für die Realisierung der Applikation.
Anschließend werden in Kapitel 6 ausgewählte Aspekte der Realisierung des Prototypen vorgestellt.
Abschließend erfolgt eine Zusammenfassung der Arbeit. Dabei wird rückblickend der aktuelle Stand der Entwicklung dargelegt und Ausblicke gegeben.

2. Grundlagen

Die Motivation zu dieser Arbeit, zeigte, dass das Dialogsystem von Alexander Bokov, Wissen in Form von Ontologien repräsentiert. Fernern wurde klar, dass die Aothoring-Software für das webbasierte Dialogssystem ein Teil des Systems selbst werden sollte. Das Dialogsystem soll also sein eigenes Administrator-Webinterface mitbringen, welches dem Autor erlaubt, einen Dialog zwischen Mensch und Maschine mit Inhalten zu füllen. Außerdem wurde festgestellt, dass die Aothoring-Software mehr leisten muss, als Eingabe von Texten in HTML Formulare. Benötigt wird zwar ein Webinterface aber gleichzeitig auch eine Desktopapplikation - eine Rich Internet Anwendung.

In diesem Abschnitt wird zunächst die Lehre der Ontologie-Wissenrepräsentation vorgestellt. Anschließend beschäftigt sich der Abschnitt mit der RIA Definition. Weiterhin werden die RIA Techniken in mehrere Kategorien unterteilt und typische Vertreter, der jeweiligen Kategorie genannt. Schließlich wird eine Entscheidung zu einer bestimmte Technik für die Editor Applikation Realisierung getroffen.

2.1. Ontologien

Um im weiteren Verlauf der Arbeit, die Funktionsweise des Dialogsystems von Alexander Bokov zu verstehen und daraus die notwendigen Anforderungen für die Authoring-Software ableiten zu können, muss ein Einblick in das Thema der Ontologie-Wissensrepräsentation gegeben werden. Diese Arbeit liefert keine Transferleistung der Ontologie Lehre auf ein Softwaresystem, denn dies ist bereits durch das Dialogsystem von Alexander Bokov erbracht worden. Daher reicht ein pragmatischer Ansatz für die Erläuterung der Theoriekonzepte aus. Einen solchen Ansatz liefert [Halder04].

Man stelle sich vor, dass das Dialogsystem, wie ein neugeborenes Kind ist. Es kann kein einziges Wort verstehen, das ein Mensch ihm sagen könnte. Jedoch ist es, wie ein Kind lernfähig und die Sprache mit der man ihm etwas beibringen kann, sowie die Form, in der es das Wissen speichert ist eine Ontologie. Um zu verstehen, wie das System lernt muss man sich klarmachen, was die grundlegenden Aufgaben und Probleme der Informationsverarbeitung sind: Nämlich bestimmte Ausschnitte der realen Welt in eine geeignete Darstellungsform zu überführen. Mit einer Ontologie kann also das Wissen über einen bestimmten Weltausschnitt in vereinfachter, abstrakter Form beschrieben und damit auch in Informationssystemen dargestellt und verarbeitet werden.

Dabei geht man folgender Massen vor:

1. Eine gemeinsame Begrifflichkeit schaffen

Diese allgemeinen Begriffe werden Konzepte oder auch Klassen genannt. Sie beschreiben eine gemeinsame Eigenschaft von konkreten Dingen und bilden somit die Sprachdomain der Maschine.

Mensch: *Es gibt Planeten.*
Maschine: Planet ist ein Konzept.

Mensch: *Es gibt Sterne.*
Maschine: Stern ist ein Konzept.

Oft können diese Konzepte weiter generalisiert werden:

Mensch: *Planeten sind feste Körper.*
Maschine: Das Konzept `Planet` leitet sich von dem Konzept `fester Körper` ab.

Mensch: *Sterne sind gasförmige Körper.*
Maschine: Das Konzept `Stern` leitet sich von dem Konzept `gasförmiger Körper` ab.

2. sMögliche Beziehungen zwischen den Begriffen definieren.
   Die möglichen Beziehungen zwischen den Begriffen werden als Relationen oder Rollen bezeichnet. Hier wird allgemein festgelegt, in welche Zusammenhänge die Begriffe überhaupt gesetzt werden können. Hierfür schafft man eine Relation und gibt an, zwischen welchen Konzepten diese bestehen kann.

   Mensch: *Planeten können sich um Sterne drehen.*
   Maschine: `Planet` kann zu `Stern` in Relation `sich drehen um` befinden.

3. Tatsachen mit Hilfe der Begriffinstanzen und tatsächlichen Beziehungen als Wissen ablegen.
   Nun, da die Maschine ein allgemeines Abbild der realen Welt, durch Konzepte und Relationen besitzt - *„Es gibt Planeten und Sterne. Planeten können sich um die Sterne drehen"* - kann konkretes Wissen abgelegt werden. Man spricht dabei von Instanziierung der Begriffe (auch erschaffen von Individuen) und bringen der Instanzen in Relation(auch annehmen der Rollen).

   a) Wissen ablegen durch konkrete Instanzen der Begriffe :

   Mensch: *Die Erde ist ein Planet.*
   Maschine: `Erde` ist eine Instanz des Konzepts `Planet`.

Mensch: *Die Sonne ist ein Stern.*
Maschine: `Sonne` ist eine Instanz des Begriffes `Stern`.

b) Wissen ablegen durch bringen der Instanzen in Relation:

Mensch: *Die Erde dreht sich um die Sonne.*
Maschine: `Erde` steht zu `Sonne` in Relation `sich drehen um`.

Zusammengefasst, also erschaffen Ontologien zunächst eine Erfahrung in Form von Konzepten und möglichen Relationen, so dass anschließend konkretes Wissen aus der Domain dieser Erfahrung, mit Hilfe von Instanzen und treten dessen in die tatsächlichen Relationen gebildet werden kann.

2.1. RIA Allgemein

Bevor zu den Rich Internet Anwendungen übergegangen wird, soll zunächst klar werden, was eine Internet Anwendung oder Webanwendung überhaupt ist. Die Wikipedia Definition lautet:

„Eine Webanwendung oder Webapplikation ist ein Computer-Programm, das auf einem Webserver ausgeführt wird, wobei eine Interaktion mit dem Benutzer über einen Webbrowser erfolgen kann. Hierzu sind der Computer des Benutzers (Client) und der des Dienstanbieters (Server) über ein Netzwerk wie das Internet oder über ein Intranet miteinander verbunden, so dass die räumliche Entfernung zwischen Client und Server unerheblich ist.“ [DeWikiWeb10]

Dies ist natürlich nur bedingt wahr und trifft gerade nicht auf RIA's vollkommen zu, da sie gerade nicht nur auf einem Webserver ausgeführt werden. Die englische Wikipedia Definition ist an dieser Stelle viel allgemeiner:

“ *In system software, a web application is an application that is accessed over a network such as the Internet or an intranet.” [EnWikiWeb10]*

Die wichtigste Eigenschaft, also ist, dass die Anwendung mindestens auf zwei über Netzwerk verteilten Rechnern ausgeführt wird - einem Server, der bestimmte Services bereitstellt und einem oder mehreren Clients, welche die Services des Servers konsumieren.

Eine Rich Internet Application ist eine besondere Webanwendung, in der das „Rich“ dem Client zugeschrieben wird. Es bedeutet also, das der Server viel mehr Programmcode an den Client

ausliefert, damit eine besonders bereichertes Benutzererlebniss garantiert wird.

Verschiedene Autoren haben versucht eine abstrakte Beschreibung dazu zu liefern, was eine RIA leisten soll. So erstellte Jeremy Allaire in seinem Buch „A next Generation Rich Client" einige abstrakte Grundsätze[All02]. Eine eher praxisangelehnter Ansatz wird auf [WebLeo08] verfolgt. Die Anforderung beziehen sich natürlich auf die Clientseite, denn dass Rich in RIA, gehört dem Client und soll genauer definiert werden. Es sollen nur solche Leitsätze aus beiden Quellen genannt werden, die auch für in dieser Arbeit erstellte Anwendung relevant sind.

Zunächst gibt es allgemeine Anforderungen an die RIA Technologien, die dem Entwickler zu Verfügung stehen sollen:

- *"Provide an efficient, high-performance Runtime for executing Code."*
  Gemeint ist die Client Laufzeit Umgebung. Dem Benutzer muss eine Laufzeitumgebung zu Verfügung stehen, welche Programmcode vom Server empfangen kann und auf Anfrage ausführt.
- *"Provide Powerfull and extensible Object Models for interactivity."*
  Die RIA's sollen mit turingkompletten mächtigen Hochsprachen entwickelt werden.
- *"Enable rapid application development through components and re-use."*
  Die RIA Technologien sollen eine Entwicklungsumgebung mit fertigen Komponententen für schnelle Entwicklung liefern, die zu eigenen wiederverwenbaren Komponenten kombiniert werden sollen
- *„Enable the use of web and data services provided by application servers."*
  Die RIA Client Umgebing soll eine Kommunikation zum Webserver ermöglichen.

Wenn die oberen Anforderungen an die RIA Technologien gegeben sind, so kommen folgende Anforderungen auf den Entwickler zu:

- *"Invoking server-side services"*
  Der Client initiiert Programmaufrufe auf dem Server und konsumiert das Ergebnis.
- *"Keeping state on the client"*
  Der RIA Client ist ein komplettes Programm. Der Benutzer kann bestimmte Daten über mehre Masken des Programms editieren. Der Fortschritt seiner Arbeit wird dabei auf dem Client verwaltet.
- *"Architekting the View"*
  Für ein reiches Benutzererlebniss müssen graphische Schnittstellen designed werden. Diese können aus verschiedenen graphischien Komponenten zusammengesetzt sein, Daten unter einander austauschen oder diese über Service Aufruferobjekte vom Server anfordern. Es

ergibt sich eine Clientseitige Architektur, die modelliert werden muss.

Mittlerweile gibt es eine Fühle von RIA Frameworks. Eine gute Auseinadersetzung mit den gegenwärtigen Methoden der RIA Entwicklung findet man in der Masterthesis von Gerhard Janisch [Jan08]. In der gegenwärtigen Arbeit sollen lediglich die RIA Frameworks grob vorgestellt und in Kategorien unterteilt werden. Somit kann man für die Anforderungen dieser Arbeit, bestimmte Kategorien schnell ausschließen.

2.2. Ajax

Der Begriff Ajax ist auf den kalifornischen Usability Berater Jesse James Garrett zurückzuführen. Er definiert in seinem Artikel [JJGar05] einen neuen Weg Webapplikationen zu entwickeln.

„Ajax isn't a technology. It's really several technologies, each flourishing in its own right, coming together in powerful new ways"

Es werden dabei mehre standardisierte Webtechniken auf eine bestimmte Weise verwendet, so dass eine RIA entsteht. Garett bescreibt diese Techniken wei folgt:

- *"standards-based presentation using HTML and CSS"*
  W3C spezifizierte Webstandards.

- *"dynamic display and interaction using the Document Object Model"*
  Dynamische Veränderung der HTML Seite auf dem Client durch Zugriffe auf das Dokument Objekt Model der Seite, durch eine standardisiere Schnittstelle.

- *"asynchronous data retrieval using XMLHttpRequest"*
  Anfrage der Daten vom Server, asynchron, also nicht Programmfluss-unterbrechend, mit Hilfe der XMLHttpRequest Schnittstelle.

- *"and JavaScript binding everything together"*
  Javascript als Programmiersprache um die obigen Techniken zu verbinden.

Alle diese Techniken werden von Haus aus von den meisten namhaften Browsern unterstützt. Eine normale Webapplikation sendet eine Anfrage an den Server, der Server erstellt je nach Anfrage eine neue HTML Seite und sendet diese an den Clientbrowser zurück. Eine Ajax Appliakation funktioniert anders. Anstatt eine Webseite zu laden, lädt der Client eine in JavaScritpt

geschriebene Ajax Engine - Programmcode der von dem Browser ausgeführt wird. Die Ajax Engine ist nun für beides verantwortlich - Rendern des Benutzerinterfaces durch Manipulation des DOM Models, also verändern des Aussehens der HTML Seite lokal auf dem Client und kommunizieren mit den Services des Webservers. Typischerweise sind Ajax Appliaktionen oft eine Mischung der beiden Verfahren, also mehrere Webseiten mit mehreren Ajax Engines.

Der Datenaustausch mit dem Server erfolgt dabei asynchron über so genannte XMLHttpRequests. Dabei handelt es sich um eine API, mit der es möglich ist, in JavaScript HTTP-Requests an einen Server zu senden, ohne dass die Seite dabei neu geladen werden muss. Dies erfolgt so, dass nach einer Benutzeraktion eine JavaScript-Funktion aufgerufen wird, die den Request versendet. In der Zeit, während nun der Browser auf die Antwort vom Server wartet, kann der Benutzer mit der Webanwendung weiterarbeiten, ohne durch die zeitraubende Kommunikation blockiert zu werden. Trifft dann später die Antwort vom Server beim Browser ein, Z.B In Form von XML-Daten, aktualisiert die Anwendung jene Bereiche der sichtbaren Seite, die von der Antwort betroffen sind, indem der DOM manipuliert wird. Alle anderen Bereiche der Seite, in der der Benutzer inzwischen eventuell Änderungen durchgeführt hat, bleiben dabei unbeeinflusst.

2.3. Ajax Komponenten Frameworks

Wegen einiger Schwierigkeiten bei der Umsetzung von Ajax Applikationen, die bei der Entscheidung zu einer geeigneten Technik zu einem späteren Zeitpunkt betrachtet werden, wurden Ajax Komponenten Frameworks entwickelt. Diese Frameworks versuchen, die Vielfalt, der Technologien, die hinter Ajax stehen, sowie das ganze webseitenorientierte Model des Internets vollkommen zu verbergen. Hierfür werden JavaScript Compiler eingesetzt, die aus einen bekannten Hochsprache JavaScript Code erzeugen. Ferner bringen diese Frameworks vorgefertigte graphischen Komponenten, die nur darauf warten mit Daten gefüttert zu werden.

Das muss sich so vorstellen, als würde man eine ganz normale GUI Bibliothek verwenden. Anstatt also zu versuchen das DOM einer HTML Seite zu manipulieren, erzeugt der Entwickler neue graphische Klassen mit Hilfe der Widgets aus der Framework Bibliothek und instanziiert diese zu Objekten, um die Widgets auf dem Bildschirm anzuzeigen. Der Frameworkcompiler übersetzt diesen Code später in Javascript und die Widgets aus der Framework Bibliothek übernehmen die DOM Manipulation. Der Entwickler muss sich darum aber nicht kümmern.

Sowohl Zero Kelvin (ZK)[ZK10] als auch Google Web Toolkit(G WT) [GWT10] liefern Benutzeroberflächensteuerelemente und Layout-Komponenten, mit denen einfacher als mit puren Webstandards in Webbrowsern lauffähige interaktive Benutzeroberflächen erstellt werden können, so dass desktopähnliches Verhalten gegeben ist. Zwischen ZK und GWT gibt es jedoch einen

Unterschied der für solche Ajax Frameworks charakteristisch ist. Während GWT clientzentriert ist, ist ZK serverzentriert. Kurz gesagt, liegt der Unterschied zwischen den beiden Ansätzen in der Entscheidung, wo die meiste Logik der Applikation platziert ist. [Abbildung 2.1]

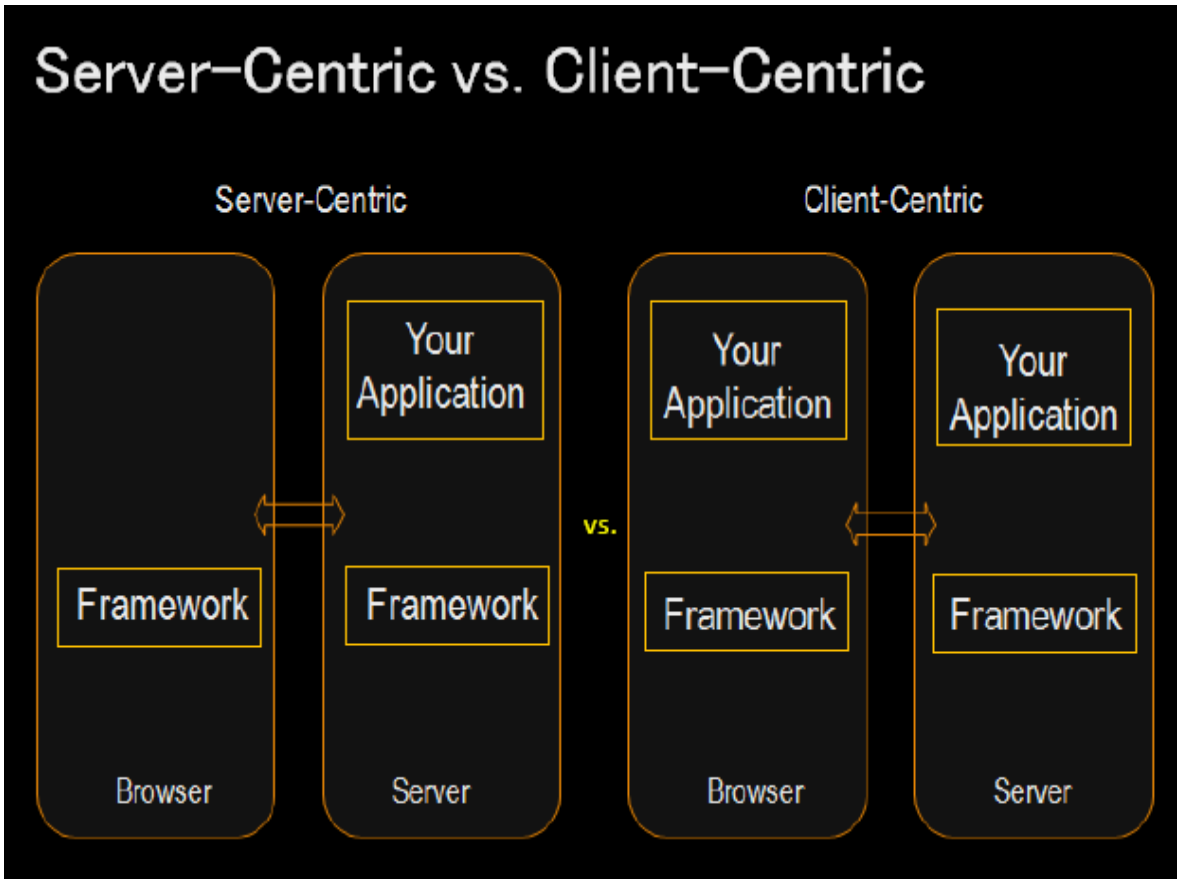


Abbildung 2.1: Server-Centric vs. Client-Centric Architecture [Liu09]

GWT macht einen klaren unterschied zwischen Clientcode und Servercode. Der Entwickler entscheidet selbst, wo er die Logik platzieren möchte. Der Server Code läuft ganz normal in der JVM ab. Der Client Code hingegen wird zu JavaScript kompiliert und läuft im Browser ab. Der Entwickler kümmert sich ferner selbst um eine Kommunikation zwischen den beiden Teilen der Anwendung. Hierfür benutz er mitgelieferte Schnittstellen, wobei er eigene Delegaten und Services von diesen ableitet[vgl. GWTRPC10] [Abbildung 2.2].

ZK hingegen abstrahiert vollkommen von dem Request Responce Lifycycle. Es entsteht vollkommen der Eindruck, als würde man eine Desktop Applikation entwickeln. Der Browser wird lediglich als Presentationsschicht der graphischen Kompinenten benutzt. D.h. nur die graphischen Komponenten werden zu HTML und JavaScript übersetzt. Java Code, der zu Behandlung von Benutzer Ereignissen verwendet wird, läuft automatisch auf dem Server ab. Die Kommunikation geschieht im Hintergrund, ohne dass sich der Entwickler darum kümmern muss.[vgl. Liu09] [Abbildung 2.3].

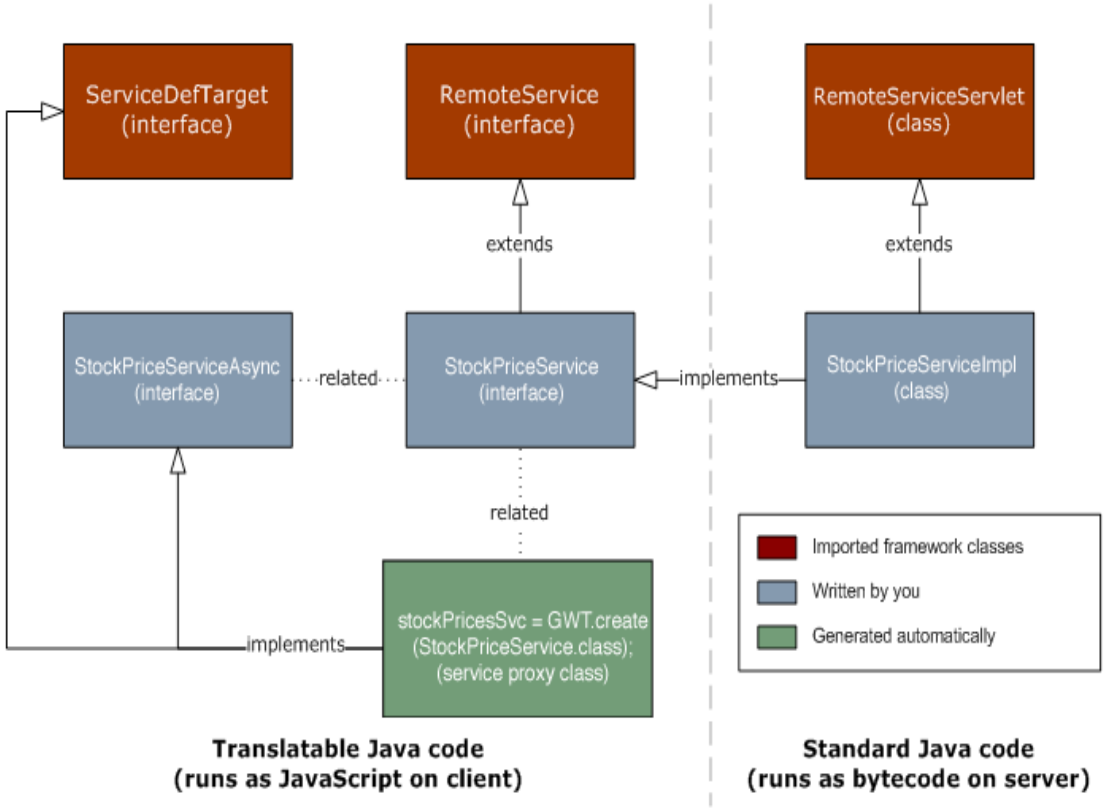


Abbildung 2.2: GWT RPC Architektur. [GWTRPC10***]***

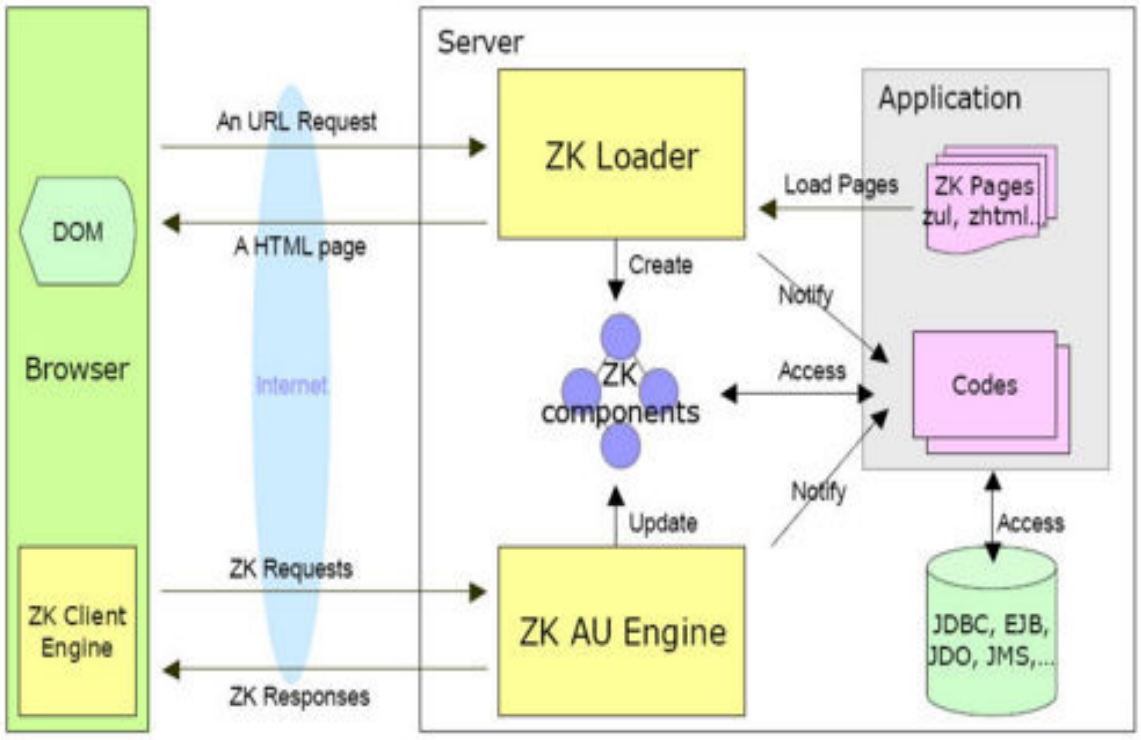


Abbildung 2.3: ZK Architektur [ZKGuide10]

2.4. Pluginbasierte Lösungen

Eine dritte Möglichkeit eine RIA zu realisieren sind Browser Plugins. Hierbei wird folgendes Konzept verfolgt:

Der RIA Anwender installiert ein Browserplugin, welches eine Laufzeitumgebung, meist eine virtuelle Maschiene mitbringt. Das Plugin selbst ist eigenständige auf dem Betreibssystem installieret Software. In den Browser wird nur Programmcode zu Integrierung dieser eingefügt. In HTML Code verlinkt der Entwickler auf eine binär kompilierte Archivdatei seines Programmcodes auf der Servermaschine. Wenn diese HTML Startseite im Browser geöffnet wird, weiß der Browser Plugin, dass das binäre Programmarchiv ausgeführt werden muss. Das graphische Rendering übernimmt nun die Pluginlaufzeitumgebung selbst mit Hilfe der Betriebsystem Schnittstellen. Es wird also nicht auf das Browser DOM Zugegriefen. Die Anzeige wird nun in das Browserfenester integriert, so dass der Eindruck entsteht, dass die geladene Applikation ein Teil der HTML Seite ist.

Bekannte Vertreter auf diesem Gebiet sind Sun's Java Applets, Adobe's Flash Player und Microsoft's Silverlight Plugin. Um eine schnelle Applikation Entwicklung zu gewährleisten liefern alle drei Hersteller Frameworks mit fertigen Benutzeroberflächenelementen mit. Sun's neuestes RIA Framework, nennt sich JavaFx, Adobe's Lösung - Flex und Microsoft erlaubt die Programmierung von RIA's für Silverlight mit Hilfe einer Untermenge von WPF (Windows Presentation Foundation). [Java10][Flex10][Silver10]

2.5. Frameworksanalyse

Es wurden grob drei Kategorien von RIA Technologien vorgestellt - Ajax, Ajax Frameworks und pluginbasierte Lösungen. Bevor Überlegungen zu Wahl einer bestimmten Technologie getan werden, kann anhand der Zielgruppe und der Anforderungen der Anwendung die Wahl auf eine Kategorie eingegrenzt werden.

2.5.1. Ajax

Es sollte bereits deutlich geworden sein, dass Ajax große Schwierigkeiten bei der Realisierung einer RIA mitbringt. Ansonsten wäre die Existenz von Ajax Frameworks überhaupt nicht gerechtfertigt. Welche Schwierigkeiten es sind, zeigt sehr deutlich Douglas Crockford, Yahoo JavaScript Chef-Architekt und Erfinder des JavaScript Serialisierungsformats JSON[Crock10]:

- Browserinkompatibilitäten und Bugs

Obwohl JavaScript durchaus eine mächtige Hochsprache ist, sind es dessen Implementierungen in Browsern nicht. Es gibt Fehler, die sich historisch in die Implementierungen angeschlichen haben und viel mehr gibt es einige Funktionen, die von einem Browser unterstützt werden und vom anderen nicht oder auf eine andere Weise. Gerade der Grundstein von Ajax, der Zugriff auf DOM und die HttpRequest Verwendung kann auch Unterschiede aufweisen. Gleiches Problem tritt auch bei den Webstandards HTML und CSS auf. Obwohl es W3C Spezifikationen für diese Standards gibt, kann gleicher Code wegen fehlerhafter Implementierungen in verschiedenen Browsern zu unterschiedlichen Layouts führen.

- Komplexe DOM API
  Die DOM API zum zu clientseitigen Manipulation der Seite ist sehr schwerfällig. Simple graphischen Manipulationen gestalten sich schwierig, da sie mit unnötigem Overhead durch den Zugriff auf die DOM API verbunden sind.

- Fehlende Template Komponenten
  Graphische Funktionalität, die über HTML Formulare hinweg geht, muss händisch implementiert werden. Hier können Funktionsbibliotheken von Dritttanbietern jedoch Abhilfe schaffen.

- Fehlende Best Practices und ungewöhnliches Programmiertparadigma
  Es gibt wenig gute Literatur, die eine Richtlinie für Ajax Anwendungen schafft. Wegen des Webseiten Paradigmas, ist die Wahrscheinlichkeit groß, dass Ajax Anwendungen, keine geschlossenen Sofware Einheiten werden, sondern architekturlose Ansammlungen von Code über mehrere HTML Seiten verteilt.

Wegen so vieler Nachteile von Ajax stellt sich überhaupt die Frage, wieso es überhaupt Anwendung findet. Die Antwort liegt in der Zielgruppenreichweite:

- Breites Publikum
  Alle Browser unterstützen die Ajax Standards. Somit muss dem Benutzer keine Installation von Drittsoftware zugemutet werden, was einige technisch nicht affine Benutzer verschrecken würde.

2.5.2. Ajax Frameworks

Ajax Frameworks versuchen die oberen Nachteile von purem Ajax zu lösen. Die Vorteile sind:

- Breites Publikum kann beibehalten werden

Ajax Frameworks übersetzen den Code in die gängigen Browserstandards - HTML, CSS, JavaScript.

- Browserinkompatibilitätten und DOM Zugriffe werden umgangen
  Die Frameworks implementieren Workarounds, so dass Code in allen Browsern trotz bekannter Bugs richtig funktioniert. Ferner muss der Entwickler keine händischen DOM Zugriffe machen, dies übernimmt das Framework.

- Wiederverwendbare Komponenten und bekanntes Programmierparadigma
  Die Ajax Frameworks sorgen für eine Architektur in der Anwendung, um Logik von Design zu trennen und bringen viele Benutzeroberflächensteuerelemente, unter Anderem mit ausgefallenen graphischen Verhalten mit. Diese Elemente sind die Bausteine für eigene customisierte wiederverwendbare graphische Klassen.

- Evt. Abstraktion von dem Request Repsponce Lyficycle
  Bei Serverzentrierten Frameworks wie ZK besteht die größte Ähnlichkeit zu dem bekannten Entwicklungsmechanismus von normalen Anwendungen. Overhead für Kommunikation entfällt.[1]

Nun scheint es so, als ob die Ajax Frameworks die Lösung sind, da sie alle Nachteile von Ajax eleminieren, aber erlauben nur mit Webstandards für ein breites Publikum zu entwickeln. Dennoch bringt der Einstatz dieser Frameworks ein gewisses Risiko:

- Erschwernis bei der Erstellung eigener Komponenten.
  Falls es passieren sollte, dass die Standardwidgets des Frameworks, die gewünschte Funktionalität nicht abdecken, wird es in einem Framework doppelt so schwer als mit puren Ajax Techniken, eigene Komponenten mit Sonderverhalten zu implementieren. Denn neben dem ganzen Aufwand und Know How für die Webstandards muss auch eine Einbindung deren in das Framework geleistet werden, wofür eben zusätzlich eine genaue Kenntniss der Frameworkimplementierung erforderlich wird.

2.5.3. Pluginbasierte Lösungen

Pluginbasiete Lösungen sind die Königsklasse der RIA's. Das Plugin bringt eine eigene Laufzeitumgebung und eine graphische Renderingsengine mit. Somit ist die Anwendung von dem Browser DOM unabhägnig. Graphisch sind hier beinahe keine Grenzen gesetzt. Sogar 3D Ani-

1 Dies kann aber auch als nachteilhaft erweisen, falls die Anforderung besteht, den Server zu entlasten, indem großer Teil der Berechnungen auf den Client verlagert wird.

mationen sind theoretisch möglich [vgl. Adobe3D10]. Frameworks zu Erstellung solche pluginbasierter Anwendungen erlauben beinahe jede Software als ein Service über das Web auszuliefern. Dabei verwendet der Entwickler ihm bereits bekannte Software-Paradigmen. Als einzige Umstellung gilt lediglich, Daten vom Server anzufordern und nicht vom lokalen Rechner.

Der einzige Nachteil solcher Lösungen, ist es Vertrauen und Akzeptanz der Benutzer zu dem Plugin zu schaffen. Über die Plugininstallation muss die RIA Laufzeitumgebung in den Browser integriert werden. Es kann viele Benutzer abschrecken, wenn sie beim Besuch einer Webseite aufgefordert werden etwas zu installieren.

2.5.4. Eingrenzung der Entscheidung

Die Authoring-Software richtet sich an eine überschaubere Anzahl der Anwender. Diese müssen zwar technisch nicht affin sein, jedoch ist denen eine Installation des entsprechenden Browserplugins zuzumuten bzw. wird für sie übernommen. Auf der anderen Seite ist das Dialogsystem nicht einfach überschaubar, so dass zu dessen Darstellung, durchaus komplexere graphische Elemente denkbar sind. Wie man in der Einleitung sieht, trifft das Dialogsystem die Entscheidungen, welche Fragen es dem Benutzer stellt erst im Verlauf des Dialogs und pflegt hierfür eine Wissensbasis. Es wäre also denkbar, dass die kognitiven Verknüpfungen der Informationen aus der Wissenbasis zu den Dialogtexten, oder gar die Wissensbasis selbst graphisch abgebildet werden müssten.

Es besteht also keinerlei Grund irgendwelche unnötigen Einschränkungen der graphischen Möglichkeiten, die hohen Aufwände und die Risiken von Ajax Anwendungen hinnehmen zu müssen, so dass getrost eine pluginbasierte Lösung gewählt werden kann.

Eine bestimmte Plugintechnologie zu wählen ist jedoch keine einfache Aufgabe. Welche der zur Zeit auf dem Markt verfügbaren RIA Plugins, die beste Lösung sei, ist beinahe eine Überzeugungsfrage. Man könnte sich die Frage stellen, welches Plugin am meisten verbreitet ist, wobei Flash konkurrenzlos vorne liegen würde, da es am längsten am Markt ist und mittlerweile zum De-facto-Standard des Webs gehört. Jedoch, wie oben erwähnt, spielt diese Tatsache bedingt eine Rolle, da eben eine an interne Mitarbeiter gerichtete Software erstellt werden muss, man also nicht auf Akzeptanz des Plugins angewiesen ist und somit auch argumentieren kann, dass neuere Technologien aus den Fehlern der Konkurrenz lernen und bessere architektonisch sauberere Lösungen schaffen.

Es gibt dennoch eine Tatsache, die dem Flex Framework von Adobe, welches als Ziel-Laufzeitumgebung, das Flash Plugin benutzt, eine Priorität verschafft. Es geht dabei um die Anbindung

des Clients an das Backend.

2.5.5. Vorteil Flex

Adobe bietet die einfachste Lösung an, um eine Kommunikation zwischen dem Client, der vom Browserplugin ausgeführt wird und dem Backend auf dem Server, aufzubauen. Die Lösung beschränkt sich auf Java Backends, was für das Dialgosystem von Alexander Bokov zutrifft. Der Vorteil liegt darin, dass Adobe eine komplette Middleware unter einer OpenSource Lizens ausliefert, die nicht nur Schnittstellen im Frontend bereitstellt um Services aufzurufen, sondern auch eine komplette Java Umgebung zu Realisierung von Backend Services bereitstellt.

Die Middleware heißt BlazeDS und erlaubt einen Remoteaufruf, beinahe wie einen lokalen Aufruf auszuführen. Dabei werden Java Datentypen automatisch zu ActionScript serialisiert. Zusätzlich wird für die Serialisierung ein besonders kompaktes binäres Format verwendet, wodurch die übertragene Datenmenge gering gehalten wird [vgl. Blaze10].

Ohne eine solche Middleware, muss man sich selbst um die Implementierung eines Webservices in Java kümmern, was mit deutlich mehr Overhead sowohl in der Implementierung als auch in der Übertragung verbunden ist.

Es gibt dabei zwei Alternativen - WSDL/SOAP Webservices und REST Websservices. Microsoft zeigt, wie solche Java Webservices erstellt und in Silverlight konsumiert werden können [Silver10]. Eine Zusammenfassung der Möglichkeiten findet sich auch in den Adobe Flex Dokumentation [Flex10, Kap 5] . Die Vorgehensweise könnte für jede pluginbasierte Lösung und sogar für normale Software angewendet werden. Zusammengefasst lassen sich folgende Erkenntnisse und Nachteile im Gegensatz zu der BlazeDS Middleware ableiten:

Ein WSDL/SOAP Webservice erlaubt die gleiche Transparenz wie BlazeDS von Adobe. Man ruft Webservices, wie lokale Methoden auf und implementiert Sie durch normale Java Objekt Methoden. Ebenfalls wie in BlazeDS, werden die korrekten Datentypen garantiert. Jedoch muss im Gegensatz zu BlazeDS, bei geringsten Änderungen des Services, die WSDL Spezifikation aus den Service-Methoden und anschließend ein Proxy-Stub für den Client zum Aufruf des Webservices aus der WSDL Spezifiaktion generiert werden. Zudem ist die Datenmenge, die durch das SOAP Protokol übertragen wird, enorm groß.

Deutlich weniger Daten werden bei einem REST Webservice übertagen. Dies aber auf Kosten von Typisierung und höherem Aufwand beim Aufruf und dessen Behandlung, da es eben keine Middleware, wie in den beiden oberen Lösungen gibt. Man implementiert Alles händisch. Bei

einem REST Webservice wird, das URL Konzept verwendet. D.h, dass der Client keine Methoden eines Proxy Objekts aufruft, sondern den Request an eine URL sendet. Es müssen also die Aufrufe der richtigen URL's auf der Clientseite verwaltet werden und auf der Serverseite richitg entgegengenommen werden. Als Übertragunsformat wird oft XML oder das etwas leichtgewichtigeres JSON Format verwendet. Beides sind Texformate und deswegen ist die Übertragungsmenge immer noch größer als in BlazeDS. Außerdem sind beide typenlos, d.h. das Übertragunsdatenformat wird nicht durch den jeweiligen Service festgelegt. Der Entwickler muss, im Gegensatz zu den vorherigen Lösungen, sich selbst um eine Serialisierung kümmern. Hierzu stehen natürlich Funtkionsbibliotheken zu Verfügung, aber er muss eben selbst je nach Service wissen, wie er die Daten korrekt interpretiert.

BlazeDS ist also eine sehr schnelle und einfache Lösung eine Kommunkination zwischen dem RIA Flex Client und dem Java Backend zu realisieren. Aus diesem Grund wird für die Realisierung dieser Arbeit das Flex Framework ausgewählt.

2.6. Adobe Flex

Eine sehr gute Übersicht über das Flex Framework findet sich in der Masterarbeit [Mar08]. Hier sollen lediglich die wichtigsten Eigenschaften genannt werden. Das Flex Framework sieht vor, dass Client Anwendungen in zwei Programmiersprachen entwickelt werden - MXML und ActionScript.

2.6.1. MXML

MXML (Macromedia Extended Markup Language) ist eine XML-basierte Auszeichnungssprache, die zur deklarativen Gestaltung von Anwendungsoberflächen innerhalb von Flex entwickelt wurde. Eine Flex Komponente wird dabei durch einen Tag mit einer Menge von, teilweise optionalen, Attributen repräsentiert.

Beispiel für MXML Code:

```
<?xml version =" 1.0„ encoding =" utf -8"?>
<mx : Application xmlns:mx=" http://www.adobe.com/2006/mxml"
       layout ="absolute">
<mx : Button id ="hello"  label ="Hello"
       x ="10" y ="40"
       click ="output.text ="Hello World" / >
<mx:Label id ="output" x ="10" y ="10" / >
```

```
</mx : Application >
```

Während des Kompilierens von Flex Anwendungen werden zuerst alle MXML- Dokumente in ActionScript transformiert. Anschließend wird daraus die binäre SWF-Datei erzeugt. Diese wird von dem Flashplayer des Benutzers geladen und somit die Anwendung ausgeführt. MXML ist also eine andere Repräsentationsform von ActionScript. MXML Komponenten bilden dabei ActionScript-Klassen ab. Eine Flex Anwendung kann also mit ActionScript und ohne die Verwendung von MXML erstellt werden. Dennoch ist die Verwendung von MXML sinnvoll:

- Die Entwicklung eines Interfaces ist innerhalb von MXML übersichtlicher als die programmatische Erstellung mittels Objekten un deren Platzierung in der Display List des Flash Players.

- Durch die Verwendung von MXML werden bereits User Interface Komponenten von der Programmlogik getrennt und so intuitiv ein bessere Strukturierung und Wartbarkeit des Quellcodes erreicht.

- Die Entwicklung von MXML-Code ist schneller als die von ActionScript-Code.

2.6.2. ActionScript 3.0

ActionScript 3.0 bildet eine objektorientierte und streng typisierte Programmiersprache mit einem Focus auf Anwendungsentwicklung und Performance. ActionScript 3.0 besteht aus zwei Teilen. Diese sind zum einen der Sprachkern und zum anderen die Flash Player API. Der Sprachkern umfasst Sprachkonzepte wie Bedingungen, Schleifen oder Typen. Die Flash Player API umfasst verschiedene Klassen zum Zugriff auf Flash-Player spezifische Funktionen, wie z.B. Die Display List oder die Benutzer Event API.

Beispiel für eine Klasse in ActionScript 3.0:

```
package {
    public class Student extends Person {
        public var matriculation_nr:int ;
        public function Student ( name:String, firstname:String ,
                                  matriculation_nr:int ){
           super(name, firstname);
```

```
            this.matriculation_nr = matriculation_nr;
        }
         public function toString ():String {
              return "Name:  " + name + " Vorname: " + firstname+ " Matrikel Nr:  " +
                      matriculation_nr;
        }
    }
}
```

Während also die Anwendungsoberflächen typischerweise in MXML erstellt werden, wird die Logik, der Anwendung, die überlicherweise aus nach einer Benutzerinteraktion ausgeführt wird in ActionSctipt erstellt. Oft wird dabei das DataBinding Feature verwendet. Der folgende Abschnitt zeigt dieses Zusammenspiel.

2.6.3. Databinding

ActionScript Code kann durch das `Script` Tag in MXML Komponenten eingebettet werden. Da eine MXML Komponente, letzendlich zu einer ActionScipt Klasse übersetzt wird, hat sowohl der ActionScript Code, als auch der MXML Code einer Komponente den gleichen Scope. Das bedeutet, dass im ActioScript Codebereich einer MXML Komponente auf alle Widgets der Komponente über deren ID Attribut zugegriffen werden kann. Andererseits sind alle Instanzvariablen des ActionScript Block gleichzeitig Instanzvaribln der gesamten Komponente und somit auch in MXML sichtbar.

Flex bringt ein Feature mit, welches erlaubt die Daten von einer bestimmten Quelle, bei Änderung automatisch in ein bestimmtes Ziel zu übertragen. Es heißt DataBinding und wird meistens dazu verwendet, um Daten und dessen Presentation zu trennen. Bei Änderung der Daten, aktualisieren die angebunden Widgets automatisch die Anzeige. Eine Vorgensweise hierbei ist es, die Daten als Binding-Quelle durch das `[Bindable]` Metatag zu markieren und bestimmte Attribute der Widgets als BindingZiel.

Das Beispiel unten zeigt die Einbettung von ActionScript in MXML und die Verwendung von DataBinding:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"  >
<mx:Script><![CDATA[
    import mx.collections.ArrayCollection;
```

```
    //Markiere die Instanzvariable students als DataBinding Quelle
    [Bindable]
    public var students:ArrayCollection = new ArrayCollection;

       public function createStundent(){
              //lese text der Input Widgets
              //und erzeuge einen neuen Studenten aus der Eingabe
              var student:Student = new Student(
                     name_input.text,
                     firstname_input.text,
                     matriculation_nr_input.text
              );

             //Speichere den neuen Studenten in der Instanzvaribale students
              this.students.addItem(student);
       }
]]></mx:Script>
  <!-- Inputtext für Name, Vorname, MatrikelNr   -->
  <mx:TextInput id="name_input" />
  <mx:TextInput id="firstname_input" />
  <mx:TextInput id="matriculation_nr_input" />

  <!-- Button zum Einfügen des neun Studenten,
       mit den eingebenen Daten aus den Textinputs
  -->
  <mx:Button label="Einfügen" click="createStudent()" />

  <!-- Erzeuge eine Tabelle von Studenten,
       die als DataBinding Ziel der students Collection gilt.
   -->
  <mx:DataGrid dataProvider="{students}" />

</mx:Application>
```

Mit geringer Erweiterung des Codes für das Layout der Komponenten würde, dieses Appliaktion wie in der [Abbildung 2.4.] dargestellt aussehen. Der Benutzer gibt Namen, Voranamen und MatrikelNr des Studenten ein, drückt den "Einfügen" Knopf und fügt somit den Studenten in die

untere Tabelle ein.

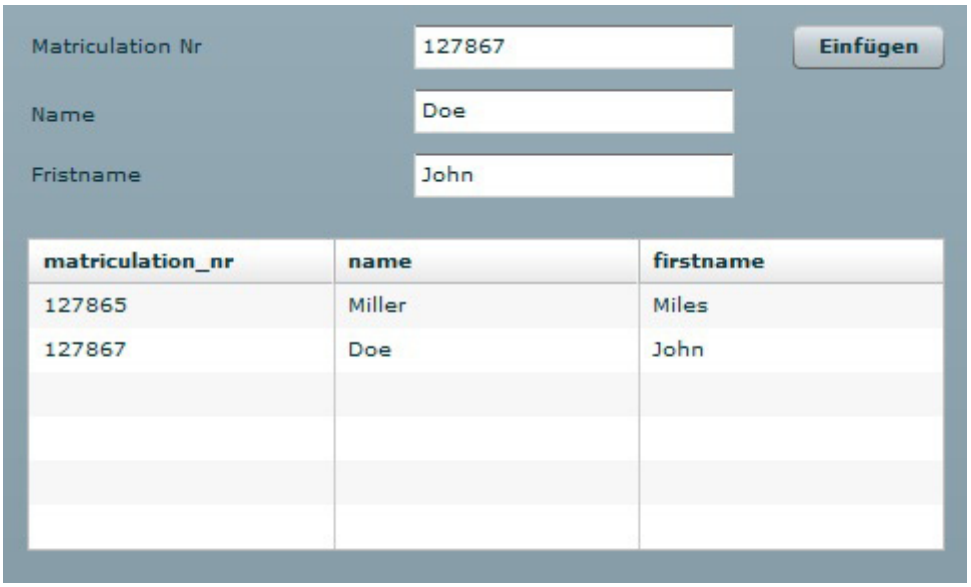


Abbildung 2.4: Datbinding Beispielapplikation

Zu Behandlung des Clicks auf den Button wird ein sog. MXML InlineEventListener verwendet - `<mx:Button ... click="createStudent()"`. Dies sorgt dafür, dass beim Click auf den Button, die in dem `Script` Tag spezifizierte Methode der Komponente - `createStudent()` – aufgerufen wird. Wie man sehen kann, hat kann innerhalb der Methode auf den Zustand der Input Widgets zugegriefen werden. Dabei wird der eingegebene Text aus den Inputs gelesen und daraus ein neuer Student erzeugt und in die `students` Collection eingefügt. Nun kommt das DataBinding zum tragen. Die `students` Collection wird mit dem `[Bindable]` Metatag als eine Binding Quelle markiert. D.h., dass nun Komponenten auf die Änderung in der Collection reagieren können. Die Studententabelle (`DataGrid`) soll nun genau dies tun. Sie repräsentiert den Zustand in der Collection. Hierfür wird deren Attribut `dataProvider` als Binding Ziel der Studenten Collection mit Hilfe von geschweiften Klammern Syntax markiert - `<mx:DataGrid dataProvider="{students}" />`.

DataBinding ist natürlich keine Magie. Beim Kompilieren fügt der Compiler zusätzlichen Code hinzu, so dass beim Änderung mit [Bindable] markierter Variablen ein PropertyChangedEvent ausgelöst, wird und die als Ziel spezifizierte Komponente darauf reagiert. [vgl. Flex08, Kap 40]

2.7. Schlusswort

Das Grundlagen Kapitel lieferte einen Basiseinstieg in das Thema Ontologien und RIA's. Schließlich wurden das Flex Framework mit dem Target Flash Player Plugin für die Realisierung der

Editorapplikation ausgewählt. Der folgende Abschnitt analysiert das Dialogsystem selbst und Leiter her, an welchen Stellen es konfiguriert wird.

3. Analyse

Das Kapitel Analyse nimmt sich als Ziel vor, die Problematiken für die Erstellung der Authoring-Software hinsichtlich einer klaren Definition der Anforderungen an den Autor zu betrachten. Bevor jedoch klare Anforderungen aufgestellt werden können, muss die Funktionsweise des Dialogsystems verstanden werden. Hierzu wird zunächst ein kurzer möglicher Dialog zwischen Mensch und Maschine vorgestellt, an dem die Lernfähigkeit der Maschine angedeutet wird. Danach wird die Funktionsweise ein wenig genauer betrachtet. Nach dem Betrachten dieses groben Aufbau des Dialogsystems wird gezeigt welches Ergebniss ein Editor liefern müsste, damit das Beispiel aus der Einleitung korrekt funktioniert. Diese grobe Analyse und das beispielhafte Ergebniss liefern genügend Anwendungsfälle um schließlich genaue Requirements an den Editor und somit an den Autor-Experten, der ihn bedient zu definieren.

3.1. Einleitung

Als Einstieg in das komplexe Thema des lernfähigen Dialogs zwischen Computer und Mensch soll ein kleines Szenario sein. Dieses mag etwas simpel, vorkommen, aber es wird dennoch genügend Stoff bieten um im weiteren Verlauf alle Konfigurationsmöglichkeiten des Dialogsystems erläutern zu können.

Der Benutzer gibt die Webadresse des Dialogs ein und gelangt auf die erste Seite, wobei ihm folgendes Bild Präsentiert wird.

Maske 1.

```
Herzlich Willkommen zu einem Dialog zwischen Mensch und Maschine. Fangen wir doch
mal mit dem Thema Weltbild an.

Jeder hat ein anderes Weltbild-Verständnis und dieses Thema beschäftigt die Mensch-
heit seit Anbeginn der Zeit....

Woran glauben Sie im Leben?
a) exakte Wissenschaft
b) Religion
c) Magie
```

Tatsächlich wird die Aufgabe des Autors sein genau solche Masken zu erstellen. Alle diese Di-

alogseiten, welche der Benutzer später in seinem Internetbrowser sieht, haben inhaltlich exakt den gleichen Aufbau.

Zunächst gibt es eine kurze Einleitung, die das Thema der Seite vorstellt. (`Herzlich Willkommen ..`). Diese Einleitung soll ab nun mit dem Terminus Preambel bezeichnet werden. Auf die Preambel folgt nun ein längere informativer Text, der in diesem Beispiel eher kurz ausgefallen ist. (`Jeder hat ein anderes Weltbild ...`). Nach dieser Information wird anschließend allgemein gesagt eine Frage gestellt (`Woran glauben Sie im Leben?`) und der Benutzer erhält Auswahlmöglichkeiten (`a) Magie, b) Religion`), mit denen er die Frage beantworten kann. Diese Frage wird ab nun Challenge[2] bezeichnet und die Antworten zu englisch Responces. Die Entscheidung für eine Response führt zu einer weiteren Maske mit gleichen Aufbau.

Die Aufgabe des Autors besteht nun darin viele solche Dialogmasken zu erstellen. Die Motivation liegt bei diesem Dialog natürlich dabei ein möglichst menschliches Verhalten zu simulieren. Es dürfen also nicht willkürlich alle möglichen Themen besprochen werden. Das System soll auf die Antworten des Benutzers eingehen und grob formuliert, nur solche Themen ansprechen, die den Benutzer auch interessieren.

Wie das aussehen könnte, kann am Besten beispielhaft in zwei weiteren Schritten erläutern:
In der weiteren Dialogseite, soll nun das System vergenauen, wie sehr das ausgewählte Thema den Benutzer beschäftigt.

Maske 2

```
.... (Preambel)


.....
.....(Info)
.....

Wie sehr beschäftigt Sie das Thema X?

a) sehr
b) eher mäßig
```

X soll hier natürlich durch das zuvor ausgewählte Thema (Wissenschaft, Magie oder Religion) ersetzt werden.

2 Der englische Begriff Challgenge ist an dieser Stelle viel treffender. Tatsächlich muss die Challenge nicht als eine Frage formuliert werden, sondern eher als eine Aufforderung zu einer Entscheidung aud die gegeben Möglichkeiten. Das Bild Frage Antwort reicht jedoch für das allgemeine Verständnis aus.

Beantwort der Benutzer die Frage mit „sehr“, so wird das System sich weiter genauer mit dem Thema Weltbild beschäftigen, in dem es weitere Unterthemen der ausgewählten Weltanschauung betrachtet.[3]

Angenommen der Benutzer hätte sich für Wissenschaft entschieden, so sieht die Maske 3 folgendermaßen aus:

```
...
Erfahren Sie mehr über
a) Relativitätstheorie
b) Stringtheorie
c) ...
```

Hat er sich hingegen für Religion entschieden würde die Maske 3 ganz andere Themen vorschlagen .

```
...
Erfahren Sie mehr über
a) Buddhismus
b) Hinduismus
c) ...
```

Wie aber kriegt man es hin, dass genaue dieser Dialogverlauf gewährleistet ist. Die simpelste Lösung ist, man legt eine statische Reihenfolge fest. D.h. auf der Seite 1 für die Antwort a) folgt die Seite 2, für die Antwort b) Seite 3 u.s.w. Dieser Ansatz soll als feste Verpointerung benannt werden.

Es liegt auf der Hand, dass bei dieser Vorgehensweise es eher schwierig wird, auf vorherige Entscheidungen einzugehen, Themen wieder aufzugreifen, Entscheidungen zu hinterfragen, eben all das, was in einem normalen Dialog stattfindet.

Das Dialogsystem wendet sich von dem Ansatz der festen Verpointerung nicht ganz ab. Dabei wird jedoch eine ganz andere Leitlinie verfolgt. Tatsächlich sind alle Seiten immer noch miteinander verbunden und eine Antwort führt immer zu der festgelegten Seite. Jedoch entscheidet das System selbst, ob diesen Verbindungen gefolgt wird oder nicht. Dies wird ganz einfach da-

3 Im realen Anwendungsfall würde man natürlich vor der Maske 2 das in 1 gewählte diskutieren, bevor es zu so einem Themawechsel kommt. Für die Anforderungen an die Autoringsoftware ist aber eben genau so ein Themawechsel charakhteristisch.

durch erreicht, ob eine bestimmte Anwortmöglichkeit präsentiert wird oder nicht. Somit kann man Querverbindungen zu bestimmten Themen an mehreren Stellen schaffen.

Allgemein kann man den Ansatz folgender Maßen erläutern. Es wird versucht Dialogseiten durch bestimmte Themenschlüsselwörter zu kennzeichnen oder wie man auch sagt zu annotieren. Ferner wird versucht, durch bestimmte Techniken, mit jeder gegebenen Benutzerantwort, dem System die aus der Antwort gewonnene Erkenntniss mitzuteilen. Das System sammelt die Erkenntnisse und kann selbst eintscheiden ob bestimmte Verbindungen gefolgt werden oder nicht. [4]

Dieses ist das allgemeine Prinzip, wie das Dialogsystem funktioniert. Dafür, dass diese beschriebene Wissensdatenbank gefüllt wird und die Schlussfolgerungen aus diesem Wissen korrekt gebildet werden sorgt natürlich der Autor, der den Dialog erstellt. Er muss die Dialogseiten in einem definierten Format ablegen und diese mit für die Maschine lesbaren Ausdrücken annotieren.

Ohne Software Unterstützung wird es aber äußerst schwierig sein, das definierte Format korrekt zu erzeugen und viel wichtiger noch einen Überblick darüber zu behalten, wie der Dialogverlauf tatsächlich aussieht. Diesem entgegen zu wirken ist die Kernanforderung an die Authoring-Software.

Der nächste Abschnitt spezifiziert das Datenformat für die Wissensbasis des Dialogsystems genauer und zeigt wie die einzelnen Dialogseiten verbunden werden.

3.2. Funktionsweise des Dialogsystems

Der vorherige Abschnitt erläuterte den Dialogaufbau und zeigte, dass das Dialogsystem neben den Inhaltlichen Seiten eine Wissensbasis pflegt und die Dialogseiten so annotiert werden, dass dieses Auswirkungen auf die Wissensbasis hat. Dieser Aufbau soll nun genauer betrachtet werden.

3.2.1 Funktionsweise allgemein

Die Wissensbasis des Dialogsystems kann man in drei Bestandteile gliedern. Sie sollen mit folgenden Begriffen benannt werden:

- TBox

4 Zusätzlich gibt es weitere Mechanismen um das Aussehen einer Seite interessenspezifisch zu gestalten, die obige Erläuterung ist jedoch für das allgemeine Verständnis ausreichend.

- ABox
- Persistenzschicht

3.2.1.1. Tbox

Die TBox repräsentiert ein Abbild der realen Welt. Hier werden mithilfe von einer Ontolgie die immer geltenden Tatsachen modelliert. Zwei Partner können nur in ein Dialog treten wenn sie sich über die Begriffe, die Sie in diesem Dialog verwenden einig sind. Wenn Menschen aus einem Dialog weiterhin Schlussfolgerungen ziehen, so können Sie dieses ebenfalls nur tun, weil sie im Verlauf des Lebens es gelernt haben. Die gleiche Erfahrung muss ebenfalls für den Computer geschaffen werden. Die TBox ist also analog mit der menschlichen Erfahrung zu setzen.

Für das Beispiel von oben, welches sich dem Thema Weltbild widmet bedeutet es also folgendes: Die TBox muss beinhalten, dass ein Begriff Weltbild existiert. Außerdem steht hier, dass Magie, Religion und Wissenschaft verschieden Inerpepretationen des Wildbilds sind. Ferner gliedern sich diese Begriffe in weitere Unterthemen - Relativitätstheorie, Stringtheorie, Buddhismus u.s.w. Und schließlich müssen natürlich die Arten von Schlussfolgerungen definiert werden, die der Computer bilden kann. In obigem Fall würden diese wie folgt lauten: „Dialogseiten können sich einem Thema widmen“, „Benutzer können sich für ein Thema interessieren“, „Das Interesse kann groß sein“. Für drei Schlussfolgerungen muss man natürlich noch die Begriffe „Interesse“,„Benutzer“ und „Dialogseite“ überhaupt einführen.

Die einfache Vorstellung des Dialogsystems sieht vor, dass diese Begriffswelt vor der Dialogerstellung von einem Entwickler erschaffen wird. Hierfür existieren bereits mächtige Tools, wie Z.B Protege, welches unter Anderem als eine Webanwendung vorhanden ist. Zu der Aufgabe des Autors wird es gehören lediglich einen Bezug auf diese Begriffswelt, also auf die TBox, bei der Erstellung der Dialogannotation zu nehmen. Wie, wird im weiteren Verlauf deutlicher.

3.2.1.2. ABox

Die ABox kann man ganz einfach mit einem Vergleich zu TBox erklären. Wenn in der TBox allgemeine Aussagen stehen, die immer gelten, so repräsentiert die Abox die Erkenntnisse welche das System für den konkreten Benutzer anhand seiner Antworten und mithilfe der Erfahrung aus der TBox schließt. Wenn also in der Tbox definiert ist, dass Benutzer sich für ein Thema interessieren können, so kann irgendwann für einen bestimmten Benutzer in die ABox eingetragen, dass er sich tatsächlich für Wissenschaft interessiert, wenn er so eine Antwort auswählt.

Es gibt also für jeden einzelnen Benutzer eine eigene ABox. Um das Dailogsystem richtig zu

konfigurieren muss aber tatsächlich nicht einmal wissen, dass es eine sog. ABox gibt, wohin die Information eingetragen wird. Man muss nur solche Ausdrücke, die die Abox versteht an die Dialogseiten annotieren. Dazu müssen diese Ausdrücke eine spezielle Syntax besitzen und wie man denken kann sich aus der Begriffswelt der TBox zusammensetzen.

3.2.1.4. Persistenzschicht

Als Persistenzschicht werden die Daten bezeichnet, welche die Inhalte des Dialogs speichern und später zu HTML Seiten für den Benutzer übersetzt oder fachlich ausgedrückt gerendert werden. Diese Daten besitzen ein spezielles Format (XML) und erlauben neben der Speicherung der Textinhalte die angesprochenen Annotationen an diese Inhalte für die ABox.

Der Ablauf zwischen einer Benutzer Antwort und dem Anzeigen der nächsten Dialogseite läuft nun wie folgt ab:

1. Der Benutzer wählt aus den vorgegebenen Möglichkeiten eine Antwort auf die gestellte Frage.
2. Die Auswahl hat zur Folge, dass aus der Persistenzschicht die nächste Dialogseite geladen wird.
3. Die zu der Seite gehörenden Annotationen werden an einen sog. Reasoner, die Einheit, welche die TBox und die ABoxen verwaltet, gesendet.
4. Der Reasoner wertet die Annotationen aus und liefet die Auswertung an das Dialogsystem.
5. Das Dialogsystem übersetzt die (XML-)Dialogseite aus der Persistenzschicht in eine HTML Seite unter Berücksichtigung der Reasonerauswertung. D.h es entscheidet welche Textabschnitte, sowie Antwortmöglichkeiten angezeigt werden.
6. Schritte 1-5 werden wiederholt.

Zusammengefasst muss die Editor-Applikation also die Persistenzschicht anlegen und bearbeiten können. Damit der Autor die Anweisungen an die Inferenzmaschine setzen kann muss vor Allem die Ontologie aus der Tbox bekannt sein. Wie dieses im Detail aussieht, erläutert das folgende Unterkapitel.

3.2.2. Persistenzschicht und Reasoner

Der Aufbau der Persistenzschicht und die Wechselwirkung mit dem Reasoner soll nun genauer betrachtet werden.

3.2.2.1. Aufbau der Textinhalte

Zunächst haben wir gesehen, dass sich der Dialog in einzelne Seiten unterteilt. Und eine Dialogseite unterteilt sich wiederum in folgende Bereiche:

- Preambel
  Eine kurze Einleitung zum Thema der Dialogseite.
- Info
  Der wesentliche Informationstext.
- Challenge
  Eine Frage, die dem Benutzer im Anschluss an das Thema gestellt wird.
- Responces
  Die Antwortmöglichkeiten.

Die Informationseinheit zu einer Dialogseite, von Alexander Bokov als InfoBlock bezeichnet, wird jedoch anders aufgebaut als die Seiten - Präsentation auf dem Bildschirm. Der wesentliche Unterschied ist, dass ein Infoblock nicht die Responces enthält, die auf dieser Seite gezeigt werden, sonder solche, die in Folge einer Auswahl zu dieser Dialogseite führen. Um also die Maske 1 aus dem Beispiel darzustellen werden mindestens zwei InfoBlocks benötigt. Der erste InfoBlock enthält nur den Willkommenstext (`Herzlich Willkommen ...`), die Info (`Jeder hat ein anderes ...`) und die Challenge (`Woran glauben Sie im Leben?`). Die Responce zu dieser Maske (Magie, Wissenschaft, Religion) müssen sich zumindest auf einem weiteren InfoBlock befinden, der als Nachfolger referenziert wird. Denkbar wären auch mehre Nachfolger mit jeweils einer Anwort.

Diese Umstellung hängt mit der angesprochenen Tatsache zusammen, dass das Dialogsystem mit Hilfe des Reasoners die Entscheidung trifft, welche Responces und somit welche weitere Themen auf eine Challenge in Frage kämen. Hierfür werden zunächst alle potentiellen InfoBlock Nachfolger in einem InfoBlock referenziert. Somit werden alle Responces der Nachfolger auf der aktuellen Dialogseite angezeigt. Wählt der Benutzer eine Antwort, so wird der Infoblock zu dem sie gehört geladen. Wie die Auswahl der mögilchen Antworten stattfindet, wird im weiteren Verlauf deutlich.

Zunächst betrachten wir den groben InfoBlock Aufbau. Diese Anwendung arbeitet, wie bereits erwähnt, mit einfachen XML Dateien. Eine XML Datei repräsentiert genau einen Infoblock. An dieser Stelle wird bewusst einfachhalthalber auf XML Spezifikationssprachen wie DTD oder XML-Schema verzichtet. Eine genaue Spezifakation wurde vom Alexander Bokov ausgearbeitet.

```
<InfoBlock name=1>
       <Responce> Antwort A </Responce>
       <Responce> Antwort B </Responce>
       <Preambel> Hier beginnt ein neues Thema </Preambel>
       <Info> Diese Information ist sehr komplex</Info>
       <Challenge> Was möchten Sie als nächstes erfahren ?</Challenge>
       <successor-list>
              <succ> InfoBlock 2</succ>
              <succ> InfoBlock 3</succ>
              <succ> InfoBlock 4 </succ>
       </successor-list>
</InfoBlock>
```

3.2.2.2. Anweisungen an den Reasoner durch Annotionen

Es wurde bereits angesprochen, dass die Textinhalte auf einer Dialogseite in bestimmter Weise annotiert werden. Diese Annotationen werden von dem Reasoner verarbeitet. Diese Anweisungen entsprechenden den Methoden der Ontologielehre. Es ist jedoch nicht notwendig, die wissenschafliche Theorie hinter Ontologien ausführlich zu betrachten um diese Anweisungen zu verstehen. Für die Erstellung der Editor Applikation ist nur die Erkenntnis, wie viele verschiedene Typen von Anweissungen es gibt und welche Wirkung sie haben, von Relevanz.

An dieser Stelle sollte man sich lediglich an drei Termini aus der kurzen Darstellung von Ontologien aus dem Grundlagenteil erinnern: Konzepte(Begriffe), Instanzen und Relationen. Die Erläuterung der Anweisungen an die ABox wird zeigen wie diese Konzepte verwendet werden.

Der Infoblock kann 4 Typen von Anweisungen an den Reasoner enthalten:

- Nachfolger
- Facts
- Effects
- Conditions References (Condrefs)

1.Nachfolger Bestimmung (successor-list)

In der Ausführung oben, wurde bereits verdeutlicht, dass ein Infoblock mit anderen Blocks über die Nachfolgerliste verpointert wird. Dies geschieht jedoch nicht über eine einfache ID Angabe, wie in der Abbildung oben, sondern wird von dem Reasoner in die Tbox eingetragen. Genauer

gesagt werden die InfoBlock Instanzen in die Relation `followed-by` eingetragen.

```
<InfoBlock name=1>
.
.
<successor-list>
       <succ> (related InfoBlock1 InfoBlock2 followed-by)</succ>
       <succ> (related InfoBlock1 InfoBlock3 followed-by)</succ>
</successor-list>
```

Der Ausdruck `InfoBlock1 InfoBlock2 followed-by` trägt die Instanz InfoBlock1 und die Instanz InfoBlock2 in die Relatioen `followed-by` ein, was soviel heißt wie InfoBlock 1 wird von InfoBlock 2 gefolgt.

2.Facts

Jedem InfoBlock können mehrere Facts zugewiesen werden. Diese kennzeichnen den Kontext der Information, welche auf dieser Dialogseite behandelt wird, also einfach ausgedrückt das Thema des Seite.

```
<infoblock name=ib 1>
<annotation>
       <fact>(instance ib1 detailliert)</fact>
       <fact>(related ib1 Wissensachft beschreibt)</fact>
</annotation>
```

Der obige InfoBlock enthält detailierte Information, definiert sich also als Insstanz des Konzepts `detailliert` (`instance ib1 detailliert`). Außerdem beschreibt er das Thema `Wissenscahft`, bringt also sich selbst und das Thema `Wissensschaft` in die Relation `beschreibt`.

3.Effekts

Effekts sind Ereignisse, die auftreten wenn ein InfloBlock geladen wird. Es gibt 2 Arten von Effekts: Update- und Deaktivate Effekte.

a)Update Effekt

Bei einem UpdateEffekt wird eine Information in die ABox des Benutzers geschrieben.

```
<responce>
        Magie
       <effect type="update">(related Benutzer Wissenschaft interessiert)</effect>
```

```
</respnce>
```

Mit seiner Antwort „Magie“ auf die Frage „Woran glauben Sie im Leben?“ triggert der Benutzer ein Eintrag seiner `Benutzer` Instanz und der `Wissenschaft` Instanz in die Relation `interessiert`. Einfach ausgedrückt, das System merkt sich aus dieser Antwort, dass der Benutzer sich für Wissenschaft interessiert.

```
<responce>
        sehr
        <effect type="update">(instance Ineresse groß)</effect>
</respnce>
```

Der obere Effekt kennzeichnet das Interesse des Benutzers als groß, indem für in der ABox dieses Benutzers die Instanz `Interesse` dem Begriff oder fachlich das Konzept `groß` zuweist.[5]

b) Deaktivate Effekt

Ein Deaktivate Effekt ist eine Abfrage an die Inferenzmaschiene. Alle Infoblocks, dessen Fakts auf diese Abfrage matchen, werden deaktiviert. D.h. ihre Antwortmöglichkeiten (responces) werden vom System nicht mehr angezeigt, auch wenn sie irgendwo in der Nachfolgerliste referenziert werden.

```
<effect type="deactivate"> x Wissenschaft beschreibt </effect>
```

Das Thema Wissenschaft wird nicht mehr behandelt nach dem die Responce mit einer solchen Annotation gewählt wurde, denn alle Infoblocks mit dem Fakt `Wissenschaft` wurden deaktiviert und somit werden die Responces solcher deaktivierten InfoBlocks nicht auf den vorherigen Seiten dargestellt.

3.Conditon References

Beinahe jeder einzelne Infoblock Baustein kann eine sog. `condref` Annotation bekommen. Die `condref` Annotation löst eine Abfrage an den Reasoner aus. Die Abfragen selbst, da sie sich oft wiederholen, werden in einer separaten Datei gespeichert. Diese werden Conditions gennant Der Baustein mit so einer `condref` Kennzeichnung wird erst dann angezeigt, wenn die Abfrage das Resultat „wahr“ liefert. Liefert die Abfrage jedoch das Resultat „falsch“, so bekommt der Benutzer diesen Baustein nicht zu sehen.

```
<infoblock>
        <preambel>
                <para condref="wissenschaft-interesse">
```

```
                Sie interessieren sich also für Wissenschaft.
            </para>
        </preambel>
```

Entsprechend findet sich in Conditions Datei ein Eintrag für `wissenschaft-interesse`. Dieser befragt die ABox nach den eingetragenen Effekten.

```
<condition-list>
        <condition name="ausstrahlende-schmerzen">
        (related? Benutzer Wissenschaft interessiert)</condition>
</condition-list>
```

In diesem Fall wird also gefragt ob `Benutzer` und `Wissenschaft` sich in der Relation `interessiert` befinden oder mit einfachen Worten, ob sich der Benutzer für Wissenschaft interessiert. Der obere Text „Sie interessieren sich also für Wissenschaft“ wird nur bei einem bejahenden Ergebniss angezeigt.

3.2.3. Szenario

Die beiden vorherigen Kapitel haben abstrakt die Funktionsweise des Dialogssystems veranschaulicht und eine erste Vorstellung gegeben welche Informationen ein Editor pflegen müsste. Nun soll das ganze für das Beispiel aus der Einleitung umgesetzt werden.

Maske 1.

Woran glauben sie im Leben?
a) exakte Wissenschaft
b) Religion
c) Magie

Um diese Maske zu erzeugen wird zunächst ein Parent Infoblock (ib1) benötigt. Dieser hat lediglich die Challange „Woran glauben Sie im Leben?“. Um die Antwortmöglichkeiten zu definieren gibt es 2 Optionen:

1)Im `ib1` werden 3 Nachfolger (successors) definiert, wobei jeder von ihnen die entsprechende Atwort (Responce) a ,b, oder c hat.
2)Der `ib1` hat nur einen Nachfolger welcher die 3 Antworten (Responce Tags) besitzt.

Wählt man nun eine der Antworten, wird der Infoblock geladen, zu dem die Antwort gehört.

Schauen wir im weiteren die Option 2 an, denn hier können alle Typen der Anweisungen an den Reasoner durchgespielt werden. Also unabhängig von der Wahl wird immer der Nachfolger InfoBlock (`ib2`) geladen und es soll folgende Maske erscheinen:

Maske 2
Sie glauben also an X.
Wie sehr beschäftigt Sie dieses Thema?
a) sehr
b) eher mäßig

Dabei soll an Stelle von X , das Wort „Wissenschaft", „Religion" oder „Magie" stehen. Dieses wird durch eine Kombination aus Effekts und Condref markierten Tags erreicht und zwar folgender Maßen:

Jede Responce (Wissenschaft, Religion, Magie) im ib2 hat einen Update-Effekt welcher den Benutzer und einen entsprechenden Bezeichner (`Wissenschaft, Religion, Magie`) in Relation `interessiert` setzt . z.B. `related Benutzer Wissenschaft interessiert`, was so viel heißt wie – der Benutzer ist an Wissenschaft interessiert. Der Infobereich von `ib2` hat ferner 3 Textbausteine, wobei jedes mit einem Condref-Bezeichner getaggt ist. z.B `<para condref="wissenschaft-interessiert">Sie glauben also an Wissenschaft Wissenschaft</para>`.

Schließlich werden 3 Condref-Abfragen in der Conditions Datei erstellt, wo abgefragt wird ob der Benutzer in Relation mit `Religion`, `Wissenschaft` oder `Magie` über den Bezeichner (Rolle) `interessiert` steht. Das Resultat jeder Abfrage ist entweder wahr oder falsch und da nur ein Fall eintreten kann, wird nur ein Textbaustein angezeigt.

Zusätzlich wird zu jeder Responce ein weiterer Effekt vom Typ `deaktivate` abgefeuert. Dies soll sich auf den weiteren Dailogverlauf widerspiegeln. Es wird pro Antwort jeweils eine Regel folgender Art erstellt: Deaktiviere alle Infoblocks die nicht das Thema „X" (`Wissenschaft, Religion, Magie` entsprechend der Responce) behandeln, aber das Thema `Weltbild` behandeln.

Wenn der Benutzer nun die Antwort *b)eher mäßig* wählt, kann über feste Verpointerung in ein ganz anderes Gebiet verzweigt werden. Wählt er hingegen die Antwort *a) sehr*, so sollen weitere Vertiefungsthemen angeboten werden. Hierzu verpointern wir von `ib2` zu `ib3` mit der Responce *a) sehr* und zu `ib4` mit der Responce *b) eher mäßig*. `Ib4` wird also ein ganz anderes Thema behandeln, während `ib3` die Maske 3 erzeugen wird.

Maske 3

Erfahren Sie mehr über
a) Relativitätstheorie
b) Quantenphysik

Der InfoBlock `ib3`, der zum Anzeigen dieser Maske geladen wurde, hat ganz viele Pointer zu anderen InfoBlocks. Wir sehen hier aber nur solche die nicht deaktiviert wurden. Damit das Deaktivieren richtig funktioniert werden alle Nachfolger mit jeweils zwei Fact Tags annotiert. Der erste Tag setzt den InfoBlock mit dem Thema `Weltbild` in Relation, der zweite, je nach Inhalt, mit `Religion`, `Wissenschaft` oder `Magie`.

XML Aufbau der beschriebenen InfoBlocks:

```
<infoblock name=”ib1”>
	<challenge> Woran glauben Sie im Leben? </challenge>
	<successor-list>
		<succ> ib2 </succ>
	</successor-list>
</infoblock>

<infoblock name=”ib2”>
	<response>
		Relgion
		<effect type=”update”>
			Benutzer Religion interessiert
		</effect>
		<effect type=”deaktivate”>
			NOT (x Religion beschreibt) AND (x Weltbild beschreibt)
		</effect>
	</response>

	<response>
		exakte Wissenschaft
		<effect type=”update”>
			Benutzer Wissenschaft interessiert
		</effect>
		<effect type=”deaktivate”>
			NOT (x Wissenschaft beschreibt) AND (x Weltbild beschreibt)
		</effect>
```

```
        </response>

        <response>
                Magie
                <effect type="update">Benutzer Magie interessiert</effect>
                <effect type="deaktivate">
                        NOT (x Magie beschreibt) AND (x Weltbild beschreibt)
                </effect>
        </response>

        <preambel>
              <para condref="wissenschaft-interessiert">
                 Sie glauben also an Wissenschaft
              </para>.
              <para condref="religion-interessiert">
                 Sie glauben also an Religion
              </para>.
             <para condref="magie-interessiert">
               Sie glauben also an  Magie
            </para>.
</preambel>

        <challenge> Wie Sehr interessiert Sie das Thema? </challenge>

        <successor-list>
                <succ> ib3 </succ>
                <succ> ib4 </succ>
        </successor-list>

</infoblock>

<infoblock name="ib3">
<response> Sehr </response>
        <challenge> Erfahren Sie mehr über </challenge>
        <successor-list>
                <succ> ib6</succ>
                <succ> ib7</succ>
                        .....
```

```
                    .....
      </successor-list>
</infoblock>

<infoblock name="ib4">
<response> eher mäßig </response>
 .......
</infoblock>

<infoblock name="ib6">
<response> Relativitätstheorie </response>
      <fact> ib6 Weltbild beschreibt </fact>
      <fact> ib6 Wissenschaft beschreibt</fact>

      .....
</infoblock>

<infoblock name="ib7">
<response> Stringtheorie </response>
      <fact> ib7 Weltbild beschreibt </fact>
      <fact> ib7 Wissenschaft beschreibt</fact>

      .....
</infoblock>
```

3.3. Anforderungen an die Editorapplikation

Nach dem die Funktionsweise des Dialogsystems an einem Szenario verinnerlicht wurde und bereits veranschaulicht wurde welche Informationsbausteine das Dialogsystem benötigt, sollen nun diese Informationsbausteine zusammengefasst werden und die Anforderungen an die benötigte Editor Applikation klar definiert werden.

1. Infoblockhierachie anzeigen
In der Analyse haben wir die Hauptinformationseinheit Infoblock kennengelernt. Grob gesehen kann man sagen, dass die Information des Infoblocks für den Dialogsystem Benutzer zu einer HTML Seite wird. Betrachtet man den Dialog selbst, so kann man von einem baumartigen Aufbau sprechen. Es wird eine Frage gestellt, die mehrere Antwortmöglichkeiten hat. Jede Antwort

führt wieder zu einer Frage mit Antwortmöglichkeiten.

Frage 1

1)Antwort

Frage 1.1

1)Antwort

2)Antwort

3)Antwort

2)Antwort

3)Antwort

In der Analyse wurde gezeigt, dass dieser Baum der Dialogseiten sich von dem Baum der Infoblocks unterscheidet und zwar dadurch, dass nicht mehr eine Antowort (Responce) den Nachfolger referenziert, sondern die InfoBlocks sich gegenseitig referenzieren. Alle Responce Elemente eines Infoblocks tauchen im Normalfall, falls keine Anweisungen an den Reasoner erfolgt sind, auf der vorherigen HTML Seite, die durch den Vater InfoBlock enstanden ist, auf.

Wichtig ist, dass nun ein InfoBlock an mehreren Stellen referenziert werden kann. Somit wird das erzeugen redundanter Inhalte vermieden und man kann bestimmte Fragen nochmals stellen und somit die Option anbieten, ein vorher nicht ausgewähltes Thema zu nun zu behandeln.

Dieser Fakt ermöglicht die benutzerabhängige Dynamik. Ob eine Referenz tatsächlich verfolgt wird, entscheidet das Dialogsystem, welches nach und nach mit den Annotationen, die an den InfoBlock angebracht sind vervollständig wird. Durch die Mehrreferenzierbarkeit spricht man nicht mehr von einem Baum, sondern von einem Referenzengraphen. Es dürfte sich als unmöglich erweisen das dynamische Verhalten des Dailogs anhängig von den Annotationen in den InfoBlocks auf einen Blick zu visualisieren, weil man alle Kombinationen der Antowortmöglichkeiten sofort darstellen müsste. Jedoch ist das visualisieren der Vater-Nachfolger Beziehungen zwischen InfoBlocks eine machbare Aufgabe und bildet somit das erste Requirement.

2. Infoblocks bearbeiten

Die Hauptbestandteile des Dialogssystems haben natürlich einen textuellen Charakter. Die primäre Aufgabe des Autors liegt nun darin den Inhalt des Systems anlegen zu können. Der Editor ist in seiner Hauptaufgabe ein Content Management System, dessen Aufteilung dem Experten offenbart werden muss.

Der InfoBlock - die Hauptinformationseinheit dieses Content Management Systems, welche später zu einer HTML Page geredenert wird - soll dem Autor auf einen Blick präsentiert werden. Ferner soll es ihm ermöglicht werden die Texte dieser Informationseinheit zu editieren. D.h. die

definierte Aufteilung in Preambel, Info und Challenge muss gegeben sein. Desweiteren müssen bestimmter Textabschnitte klar erkenntlich als eine Antowortmögilchkeit des Benutzers (Responces) deklariert werden, die wiederum bearbeitbar werden. (Requirement 4).

Nicht zuletzt sollte es möglich sein bestimmte Annotaionen, die im Analyse Kapitel gesehen haben an den InfoBlock hinzufügen zu können (genaueres ergeben die nachfolgenden Requirements). Und schließlich muss es möglich sein neue InfoBlocks an den Beziehungsgraphen aus Requirement 1. anhängen zu können, sowie logischerweise bestehende Komponenten daraus zu entfernen.

3. Facts bearbeiten
Facts sind das erste Attribut, welches für die Dynamik des Dialogs essentiell ist. Sie setzen eine bestimmte Informationseinheit - den InfoBlock- in ein Kontext. Ein Kontext, der nicht nur für Menschen, sonder auch für die Maschine verarbeitbar sein muss. Facts erfüllen also direkt den Kerngedanken des Symantic Webs.

Eine Dailogseite wird sich einem oder mehreren bestimmten Themen widmen. Diese Thema Zuordnung entscheidet im Verlauf des Dialogs, in Zusammenhang mit den gewonnenen Erkenntnissen aus den gegebenen Antworten des Benutzers, ob die entsprechende Dialogseite dem Benutzer präsentiert wird oder nicht.

Das Requirement lautet also, maschinenlesbare Ausdrücke erzeugen zu können, welche den InfoBlock einem Kontext zuordnen. Wir haben zwei Arten von Ausdrücken kennengelernt, die eingegeben werden können : Ontolgie Relationen wie `InfoBlock1 Wissenschaft beschreibt` (Die Instanz `InfoBlock1` steht mit der Instanz `Wisssenschaft` in Relation), sowie Ontolgie Instanzen - `Infoblock1 detaliert` (Die Instanz `InfoBlock1` gehört zum Konzept `detaliert`).

Dabei muss man natürlich bedenken, dass die Bezeichner, die in den Facts verwendet werden, in weiteren Informationseinheiten abgefragt werden. (Requirement 5 b)- DeactivateEffects). Außerdem ist durchaus oft beabsichtigt, bestimmte Bezeichner für mehrere InfoBlocks zu verwenden, da logischerweise es mehrere Dialogseiten geben kann, die sich einem bestimmten Thema widmen. Dieses Requirement umfasst also ebenfalls eine Unterstützung bei der Eingabe wörtlich korrekte Bezeichner. E.g der Editor muss wissen, dass der Bezeichner `beschreibt` oder `Wissenschaft` eingegeben werden darf.

Tatsächlich entsteht bei dieser Problematik Diskussionsbedarf im Hinblick darauf ein eindeutiges Requirement zu formulieren. Bis jetzt gehen wir davon aus, dass alle diese Bezeichner sich in irgendeiner Weise in der T-Box befinden, also in der Ontologie. Die Ontologie beinhaltet also,

wie in der Einleitung erläutert, die Erfahrung, dass z.B. InfoBlocks existiren, InfoBlock bestimmte Themen beshreiben und die möglchen Themen etwa Wissenschaft, Religion u.s.w. sein können.

Handelt man nach den Standartregeln der Ontologien, so müssen also folgende Vorbedingungen gegeben sein, damit ein Ausdruck wie `InfoBlock1 Wiisenschaft beschreibt` korrekt verarbeitet werden kann:

Die TBox muss eine sog. Rolle namens `beschreibt` besitzen. Der Terminus Rolle ist hier wie Assoziation zu verstehen. Die allgemeine Assoziation oder Rollen Defenition sagt also, dass zwei Instanzen einer Klasse diese Rolle annehmen können. Diese Rollendefinition sehe in etwa dann folgender Maßen aus: `InfoBlockKonzept Thema besschreibt`. Die Betonung liegt auf dem Wort Instanzen. Bevor wir also diese Beziehung in die ABox eintragen, müssen ebenfalls Instanzen vom Typ `InfoBlockKonzept` und `Thema` erstellt werden.

Der Ausdruck `InfoBlock1 Wiisenschaft beschreibt`, sagt, aus ontologischer Sicht betrachtet, folgendes aus: Trage die Instanz `InfoBlock1` vom Typ `InfoBlockKonzept` und die Instanz `Wissenachft` vom Typ `Thema` in die Assoziation beschreibt.

Es ist nicht notwendig die obige Erläuterung bis in letzte Detail zu verstehen. Es soll lediglich klar werden, dass bevor man einen Bezeichner wie `Wissenschaft` überhaupt verwenden kann nach der Ontologielehre einige Vorbedingungen erfüllt sein müssen, welche die Existenz dieses Bezeichners definieren.

Der Diskussionsbedarf für das Requirement ergibt sich aus einer mächtigen Funktion welche der Reasoner Racer mitbringt. Racer erlaubt nämlich eine dynamische Verwendung der Bezeichner. Kurz gesagt bedeutet es, dass zu jeder Zeit ohne irgendwelche Vorbedingungen ein Ausdruck wie `InfoBlock1 Magie beschreibt` eingegeben werden darf. Racer legt im Hintergrund bei der ersten Verwendung die nötigen Instanzen und Rollen mit Standartdatentypen an. Das System lernt quasi sofort. Das Dialogsystem erlaubt diese Funktion. Defeniert man eine TBox Ontolgoie vor der Dialogerstellung, so kann dieses Wissen zusätzlich genutzt werden. Tut man es nicht, so wird das Wissen nach und nach aus den Annotationen an die InfoBlocks aufgebaut, welche durch die Benutzerantworten geladen werden.

Handelt man also nach dem üblichen Prinzip der Ontologielehre, so bedeutet, dass ein Ontologie Erstellter nach einer Beratung mit dem Experten alle möglichen Rollen und Instanzen schon vorher modelliert hat oder würde zumindest verlangen, dass der Experte immer in Kooperation mit dem Entwickler, dass System bearbeitet, was natürlich eine Einschränkung ist. In diesem Fall lautet das Requirement, bei der Eingabe der Ausdrücke immer in der Reasoner TBox

nachzuschauen ob diese korrekt sind. Möchte man das mächtige Feature der dynamaischen Objekterzeugung von Racer verwenden und somit dem Experten Freiheiten bei der Erstellung gewährleisten, so muss der Editor in der Lage sein alle bisher eingegeben Expressions, die sich quer über die InfoBlocks verteilen zu analysieren. Die obige Anforderung der Analyse der TBox besteht dann zusätzlich.

4. Responces bearbeiten

Im Requirement 2 wurde diese Anforderung bereits angesprochen. Diese sollte durch die vorherige Analyse bereits ersichtlich sein, wird hier aber vollständigkeitshalber ausformuliert.

Die Anforderung lautet die Texte der Dialogantwortoptionen editieren zu können. Dabei sind die Antowortoptionen an einen bestimmte Infoblock anzuhängen. Somit müssen hier ebenfalls die Lösch- und Einfügeoperationen implementiert werden.

Wichtig ist bereits an dieser Stelle zu bedenken, dass die Responces weitere Annotationen besitzen. Deswegen muss jede einzelne Responce ein klar getrennten Block bilden, damit der Author in der Lage ist, die Annotationen jeder Responce entsprechend zuzuorden.

5. Effekte bearbeiten

Effektte sind ein weiteres Werkzeug der Dialogdynamik. Wenn Menschen einen Dialog miteinander führen, so sammelt die fragende Person logischerweise Informationen über sein Gegenüber aus den Antworten, die diese ihm gibt. Nichts Anderes wird hier für die Maschine ermöglicht.

Die Anforderung besteht darin zu jeder Antwort des Benutzers, also zu der Response, einen Ausdruck für die Maschine anhängen zu können. Diese Annotaion wird unmittelbar nach der gegeben Antwort, genauer gesagt, nach der ausgewählten Antwortmöglichkeit verarbeitet. Dabei ist diese Verarbeitung in den meisten Fällen zunächst ein einfaches Merken des Zusammenhangs welcher aus der gegebenen Antwort resultiert, also das Gleiche was ein Mensch mit der Information machen würde. Dieses aktualisieren der Wissensbasis, ist das zuvor erwähnte UpdateEffect.

Vorgestellt wurde eine zusätzliche Möglichkeit der Antwortverarbeitung. Der Autor kann nach einer bestimmter Anwort schließen, dass ab nun bestimmte InfoBlock Referenzen nicht mehr verfolgt werden, d.h. bestimmte Themen nicht mehr bearbeitet werden sollen. Dieses war, wie bekannt, das DeaktivateEffect.

Die beiden Effecte unterscheiden sich in ihrer Definitionsformel und somit teilt sich diese Anforderung in zwei Unteranforderungen - a) UpdateEffects und b)DeactivateEffects - auf.

a) UpdateEffects

Es wurde bereits angedeutet, dass die UpdateEffect Ausdrücke syntaktisch genau gleich aufgebaut sind wie Fact Ausdrücke. Der Unterschied bestand darin, dass hier nicht mehr ein Bezug zu einer InfoBlock Instanz aufgebaut wird, sondern beliebige Fakten eingetragen werden. Die typische Anwendung ist etwas über den Benutzer des Dialogs auszusagen.

```
Benutzer Wissenschaft interessiert
```

Man kann aber auch andere Objekte charakterisieren.

```
Interesse groß
```

Die Anforderung lautet eine beliebige Anzahl von solchen Ausdrücken einer Responce zuordnen zu können und selbstverständlich diese wiederum nachträglich bearbeiten zu können.

Wiederum besteht die gleiche Problematik wie bei den Facts. Es muss sichergestellt werden das die Effectbezeichner korrekt sind. Darf z.B. Das Wort `Interesse` oder `groß` verwendet werden? Falls man hier ebenfalls das dynamische Feature der Definitionserzeugung durch Racer nutzt, so muss sichergestellt werden, das Bezeichner konstant verwendet werden. Falls sich also der Autor entscheidet den Bezeichner `Benutzer` in einem UpdateEffect in die Wissensbasis einzutragen, muss er es zumindest an einer weiteren Stelle buchstäblich gleich verwenden und zwar dort, wo aus diesem Wissen auf den Dialogverlauf reagiert wird (DeactivateEffects und Condrefs). Denkbar wäre ebenfalls das gleiche Bezeichner an mehreren Stellen dadurch verwendet werden, weil bestimmte Objekte weiter spezifiziert werden. Z.b wenn der Benutzer außer an Wissenschaft ebenfalls an Religion interessiert ist.

Bei dieser Anforderung muss also ebenfalls ein Bezug mindestens zu Ontologie TBox oder zusätzlich zu allen eingegebenen Bezeichnern in den vorherigen UpdateEffects hergestellt werden.

b) Deaktivate Effects

Deaktivate Effects waren bekannter Weise eine spezielle Reaktion auf eine Benutzer Anwort. Hier kann der Author das System auffordern bestimmte Themen für den weiteren Dialogverlauf auszuschließen.

Nochmals erläutert bedeutet das, dass durch eine Abfrage bestimmte InfoBlocks quasi abge-

schlatet werden. D.h. obwohl der Benutzer auf eine Seite gelangt, die einen solchen deaktivierten InfoBlock als Nachfolger referenziert, werden die Responces dieses InfoBlocks nicht dargestellt. Folglich besteht keine Möglichkeit mehr zu dieser Informationsseite zu gelangen. Offensichtlich muss die Deaktivate Abfrage sich an die Themen beziehen, welche den InfoBlocks vergeben wurden und demnach an die Fact Annotationen.

Die Anforderung besteht also darin, bestimmte Anfragen zu erzeugen, welche unmittelbar nach einer gegebenen Benutzerantwort ausgeführt werden und somit direkt an ein bestimmtes Responce Element angehängt werden. Der Zweck dieser Abfrage ist es, diese InfoBlocks anhand dessen Facts zu finden, worauf sie folglich aus dem Dialogverlauf ausgefiltert werden.

An dieser Stellte betrachten wir nochmals das Deactivate Effect aus dem Beispiel:
`x Weltbild beschreibt AND  !(x Wissenschaft beschreibt)`
Deaktiviere alle Seiten über Weltbild, die nicht das Thema Wissenschaft betreffen.

Die Anforderung ist also Ausdrücke aus den Facts mittels Klammerung und den boolischen `AND` und `OR` Operatoren zu beliebig geschachtelten Formeln miteinander zu verknüpfen. Tatsächlich macht es natürlich nur Sinn, nur die Ausdrücke zu verknüpfen, die man auch in den Facts eingegeben hat. Es macht logischerweise z.B. keinen Sinn, zu versuchen InfoBlocks mit dem Thema Medizin zu deaktivieren, wenn man gar kein InfoBlock mit diesem Thema gekennzeichnet hat. Beschränkt man jedoch die Benutzer Eingabe nur auf solche bereits bakannte Ausdrücke, so würde das aber auf der anderen Seite einen starren Editierungsprozess erzwingen, bedeutet also das Programm schreibt eine Reihenfolge der Bearbeitung vor - zunächst Facts und dann DectivateEffects, was weniger elegant ist. Zumindest muss natürlich wieder eine buchstäblich korrekte Eingabe gewährleistet werden.

Zum Besserren Verständniss wurde bis jetzt ein Detail, betreffend der Notation von der DeactivateEffect Formel Ausdrücken verschwiegen. Momentan wurde in den Bespielen immer die Infix Notation verwendet. Tatsächlich aber müssen die Ausdrücke in der Prefix abgelegt werden:

Infix:
`x  Weltbild beschreibt AND  !(x Wissenschaft beschreibt)`
Prefix:
`AND (x  Weltbild beschreibt  !(x Wissenschaft beschreibt) )`

Die Prefix Notation ist für Menschen schlecht lesbar. Aus diesem Grund sollten die Ausdrücke in der Anzeige von der Prefix in die Infix Notation und beim Speichern umgekehrt überführt werden.

6. Condrefs und Conditions

Mit Condrefs und Conditions wurde eine weitere Technik zu Dialogsteuerung vorgestellt. Conditions sind genau so aufgebaut wie Deactivate Effects, mit dem Unterschied, dass sie nicht mehr nach einer Menge von Objekten fragen, sondern danach ob bestimmte Aussagen gelten oder nicht. Wenn man an einer Stelle die Information `Benutzer Magie interessiert` und an einer weiteren `Interesse groß` einträgt, so macht es Sinn irgendwann den gerenderten Textinhalt eines InfoBlocks davon abhängen zu lassen. Man fragt also den Reasoner, ob eine bestimmte Bedingung gilt und nur diesem Fall wird eine bestimmte Textpassage angezeigt.

Diese Abfragen wurden Condtions benannt und da sie oft gleich sind, wurde sie in eine separate Datei ausgelagert. Hier wurden diese Formeln jeweils zu einem treffenden Schlüsselwort zugeordnet. Jedes Element im InfoBlock kann nun dieses Schlüsselwort als eine Condref verwenden, was automatisch ein Auswerten der Condition Formel an den Reasoner beim Laden des InfoBlocks für jedes dieser Elemente bedeutet. Das Resultat muss wahr liefern, erst dann wird so ein getaggtes Element weiter gerendert.

Condref => Condition

```
großes_magie_interesse => Benutzer Magie interessiert AND Interesse groß
```

Die Anforderung umfasst also zunächst Conditions zu verwalten, welche einem eindeutigen Schlüsselwort zugeordnet werden. Die Conditions besitzen dabei die gleiche Struktur wie Deaktivate Effekts. Es gelten also hier die gleichen Anforderungen aus 5b), wobei die Formel sich jedoch aus den Kombinationen der durch UpdateEffects eingetragenen Informationen zusammensetzt. Gefordert ist natürlich wieder die Unterstützung der sinnvollen Eingabe und der Überführung zwischen den Infix und Postfix Notation. Ferner muss es natürlich möglich sein, dass Condtion Schlüsswörter den InfoBlock Bestandteilen zuzuordnen. Diese Bestandteile sind die bekannten Responce, Preambel, Info und Challenge, aber viel Häufiger besteht der Anwendungsfall einfach darin, die einzelnen Textpassagen z.B. In einem Preambel Text durch eine Condref zu annotieren, so dass sich der Preambeltext je nach Benutzerinteressen unterscheidet.

7. Richtext

Es wird natürlich oft gewünscht, dass die Informationstexte auf den Webseiten des Dailogs formatiert sind. Es gibt Absätze, manche Elemente sind fett oder kursiv bzw. farblich markiert. Es wurde zuvor erläutert, dass das Dialogsystem dafür ein bestimmtes Format bereitstellt. Dieses Format kann bestimmten Textpassagen zugeordnet werden.

Der Autor muss natürlich die Freiheit besitzen diese Formate zu vergeben. Es gibt genügend

Beispiele durch diverse Textverarbeitungswerkzeuge, wie dieses umgesetzt werden kann. Die Anforderung besteht also ein Verhalten umszusetzen, bei dem das interne Format transparent bleibt, also bei dem der Autor sofort die richtige Ausgabe sieht. Beachtet werden muss, dass das Werkzeug nur solche Formatierungen erlaubt, die das Dailogsystem auch versteht, damit diese später korrekt in HTML Code übersetzt werden.

3.4. Zusammenfassung

In diesem Kapitel wurde das Patiendialogsystem analysiert und die Anforderungen an seine Autoringsoftware und somit an diese Arbeit gebildet. Es zeigte sich, dass das Dialoginformationsystem seine Wissenbasis in drei Schichten aufteilt: die Ontolgie Erfahrung-Wissensbasis - TBox, die Ontologie bnutzerspezifische Schlussfolgerung Wissensbasis - ABox und die Persistenzschicht (Infoblocks, dessen Annotaionen Einfluss zu Laufzeit an der ABox operieren). Dabei besteht die Kernanforderung an das Authoring-Werkzeug, die Persistenschicht zu bearbeiten und den Bezug zu der TBox Ontolgie herzustellen. Dieses soll für einen Autor ohne Kenntnisse der syntaktischen Regeln des Systems ermöglicht werden. Wie das ganze aussehen könnte wird das folgende Kapitel Design veranschaulichen.

4. Design

1. Einleitung

Der Abschnitt Design nimmt sich zum Ziel einen visuellen Prototypen für die Editorapplikation zu erstellen. Gerichtet an die herauskristallisierten Anforderungen aus dem Analyse Kapitel werden graphische Masken zu dessen Realisierung erstellt und die Abläufe deren Benutzung geschildert. Die Untersuchung nimmt nicht den Anspruch ein möglichst bis in letzte Detail ausgearbeitetes und benutzerfreundliches Interface zu definieren, wie es im Rahmen der Usability Disziplin verlangt wird. Zunächst soll lediglich ein Ansatz gebildet werden die komplexen Zusammenhänge des Dialogsystems visuell überschaubar zu machen.

Durch die direkte Manipulation der graphischen Kernschnittstelle des Flashplayers ist es möglich beinahe Alles zu realisieren, was in der Computergrafik möglich ist. Im Hinblick auf einen angemessenen Zeitrahmen für diese Untersuchung wird sich der Prototyp aus den graphischen Klassen des Flexframeworks und deren Kombinationen (Custom Components) zusammensetzen.

2. Gestaltung

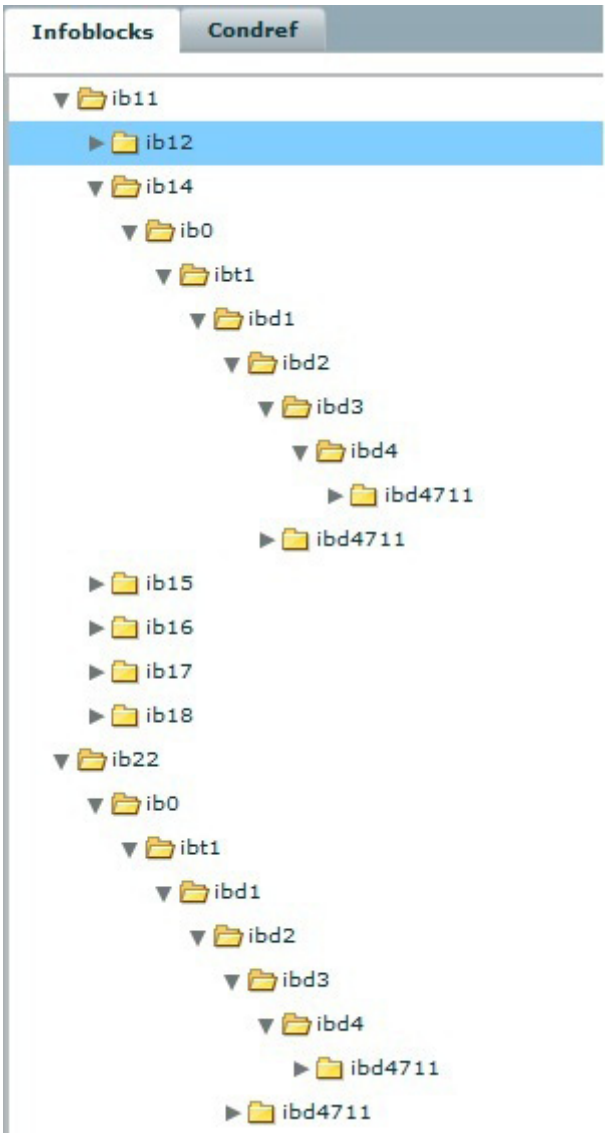
1. Infoblockhierachie anzeigen
Diese Anforderung wird der Einstieg in die Editorapplikation bilden. Dem Benutzer wird nach dem Ladebildschirm ein „Infoblockbrowser„ dargestellt. Die Anforderungsanalyse zeigte, dass alle Infoblocks sich gegenseitig referenzieren können. Es entsteht also ein Infoblockreferenzen Graph. Flex bringt leider keine Graph Klasse mit, es existieren jedoch thirdparty Implementierungen. Die geforderte Aufgabe lässt sich aber ebenfalls mit einer Baumdarstellung lösen.

Da es vorkommen kann, dass ein Knoten mehrere Male referenziert wird und es zu zyklischen Referenzen kommen kann, darf der graphische Baum nicht sofort für alle Knoten beim laden expandiert werden, sondern nur durch den Benutzer. Tatsächlich ist es fraglich ob eine Graphvisualisierung eine bessere Übersicht verschaffen würde. Außerdem liegt die Vermutung nahe, dass dieser Graph durch die Natur eines Dialogs über weite Strecken einen Baumcharakter behalten wird. [Abbildung 4.1].

Die Infoblock Hierachie aus dem Szenario in Kapitel 3 wurde in dieser Abbildung als Nachfolger von Infoblock `ibt1` eingehängt (mit `Ibd` beginnende Infoblocks). Am Beispeil von `ibt1` sieht man wie die Referenzierung an mehreren Stellen aussieht. Der Gleiche Baumabschnitt taucht exakt zwei mal auf.

Beim Auswählen der Infoblock Id in dem Baum soll der entsprechende Infoblock geladen werden. Die Erzeugung eines neuen Infoblocks soll erst außer Acht gelassen werden, da diese im Grunde dem Bearbeiten Szenario entspricht. Es sollte dabei lediglich beachtet werden, an welcher stelle im Baum neue Elemente temporär angehängt werden (gilt ebenfalls für Löschen von Vaterknoten und somit entstehende vaterlose Teilbäume). Gelöst könnte dies Problematik ganz einfach durch die Einführung eines Root Knotens , unter den alle Elemente ohne Referenz angezeigt werden. Eine konsistente Dialoganwendung muss natürlich nur einen solchen Knoten besitzen.

Abbildung 4.1: InfoBlockBrowser



2. Infoblocks Bearbeiten

Nach dem der Infoblock geladen wurde, wird er in einer separaten Maske visualisiert. Dabei kann man natürlich nicht sofort alle seine Annotationen graphisch darstellen, da das Ganze höchst unübersichtlich wäre. So lautet der Vorschlag an dieser Stelle, zuerst nur solche Elemente anzuzeigen, die dem Benutzer schnell eine Aussage über den Inhalt dieser Seite liefern, also die Responces, Preambel, Info und Challenge.

Möchte der Benutzer sich schnell ein Überblick über die Annotationen verschaffen, so wäre eine simple Texdarstellung als Vorschau denkbar.

Die [Abbildung 4.2] zeigt den Infoblock `ibd2` aus dem Beispiel Szenario, welches 3 Responces mit Entsprechenden Effekten besitzt (Effekte noch nicht dargestellt). Die mit „|>“ Gekennzeichneten Knöpfe öffnen je nach Kontext des Elements eine Auswahlliste zum weiteren Bearbeiten der Elementannotationen oder Kinderelemente. Für eine Responce müssen der Text, die Condref und die zwei Effekttypen bearbeitet werden, während für den Infoblock selbst die Facts und der Text und für Preambel, Info und Challenge nur der Text bearbeitet wird. Die weitere Bearbeitungsmaske

öffnet sich in einem Popup Fenster.

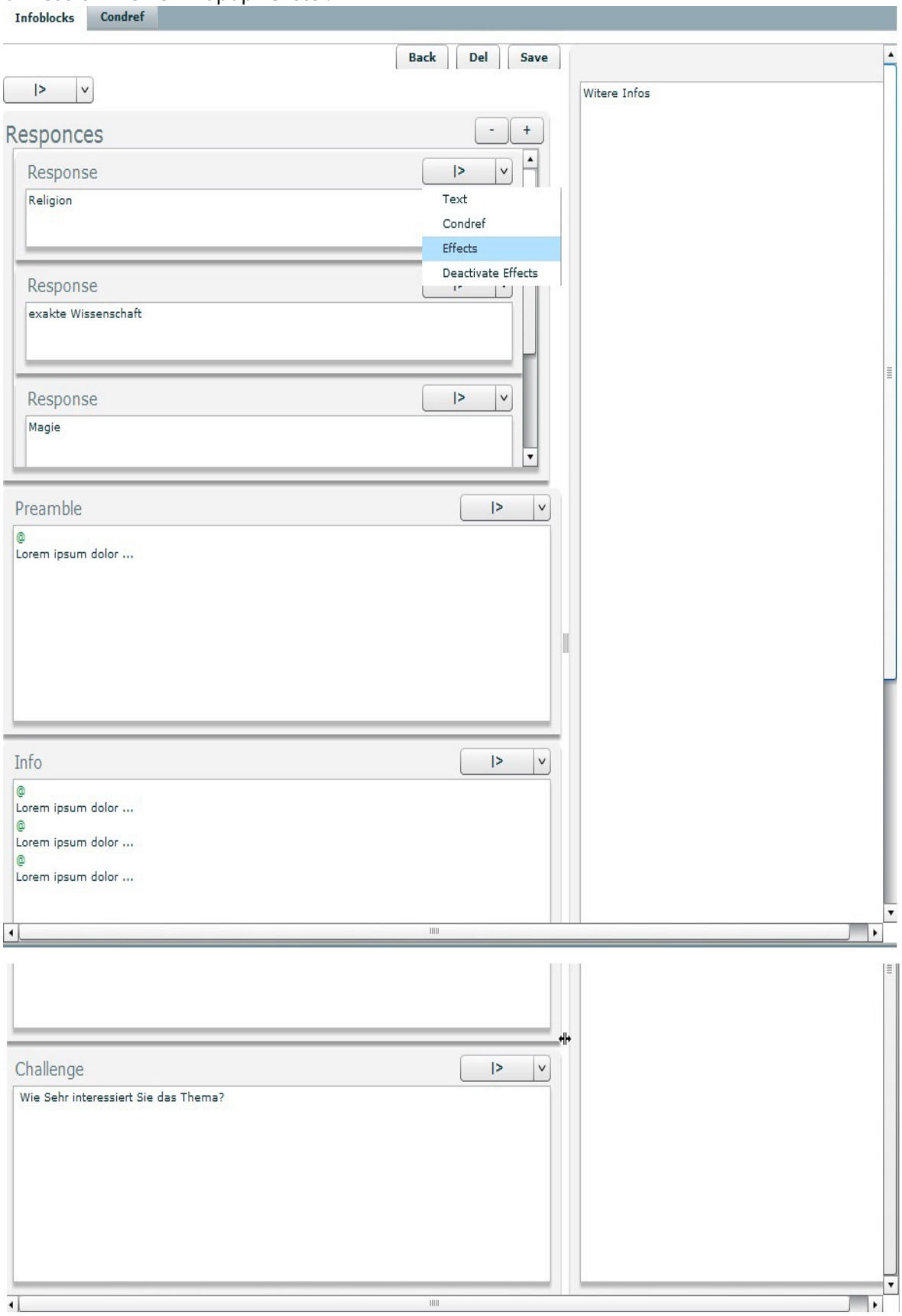


Abbildung 4.2: InfoBlockForm

3. Facts Bearbeiten

Um einem Infoblock bestimmte Fakt-Annoationen wie `related ib4 Wissenschaft beschreibt` aus dem obigen Beispiel zu vergeben, benötigt man einen Bezug zu der Ontologie, wo defeniert ist, dass ein Objekt wie `Wissenschaft` oder die Relation `beschreibt` überhaupt existiert. Natürlich kann man an dieser Stelle sich an die existierenden Tools (Racer Administration Oberfläche, Protege) richten. Diese stellen die Ontologie Datenbasis in Baumstrukturen da. Diese Strukturen sind jedoch eher für eine Gesamtübersicht geeignet und nicht für die schnelle Formeleingabe.

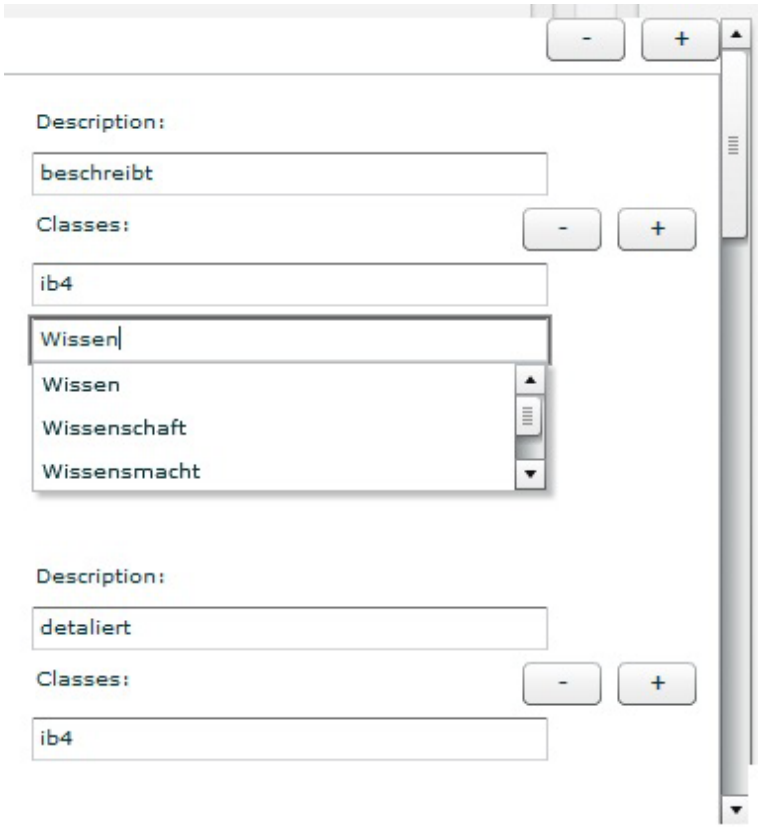


Abbildung 4.3: ExpressionsEditor

Für die Formeleingabe reichen vorerst simple Textformulare mit Autovervollständigung. Eine Visualisierung der Ontologien, aus der Bezeichner per Drag- und Drop oder Copy und Paste Übernohmen werden, wäre in Zukunft als erweiterte Formeleingabe durchaus denkbar.

In der Analyse haben wir gesehen, dass es an dieser Stelle zwei Typen von Formeln gibt: Relationen mit 3 Bezeichnern (siehe oben) oder Instanz Definitionen wie `instance ib4 detaliert` Für den Benutzer soll es transparent bleiben ob er eine Relation oder Instanz Klassifizierung eingibt. Er bekommt ein Maske zu sehen in die ein Fact Bezeichner und wahlweise zwei oder ein Objektbezeichner eingetragen werden sollen. Abhängig von dieser Wahl wird die entsprechende Formel erstellt.

Die [Abbildung 4.3] veranschaulicht die Eingabe der beiden Formelarten. Durch die + und - Zeichen im oberen Leisten Bereich, können Facts hinzugefügt und entfernt werden. Die gleichen Zeigen neben dem Stichwort `Classes` ermöglichen das Wählen zwischen zwei oder einem Objektbezeichner wie bereits erläutert.

4. Responces bearbeiten

Die Infoblock Maske zeigt die Responces mit einer Textvorschau direkt beim Laden des Infoblocks an. Das Erstellen und Löschen dieser Elemente kann also direkt in der InfoBlockform geschähen und ist trivial. Viel interessanter ist das bearbeiten von den Effects, welche an die Responces anotiert werden.

5. Update Effects

Die Analyse hat gezeigt, dass die Update Effects exakt gleiche Formeln besitzen wie Facts, also Relationen und Objekt Klassifizierungen. Der Unterschied liegt wie bereits erläutert darin , dass bei den Facts ein Object, der Infoblock selbst war und in diesem Fall beliebige Zusammenhänge eingetragen werden können.

E.g `related Benutzer Magie interessiert`

`instance Benutzer abergläubig`

Somit kann dieses Requirement mit der gleichen Maske, wie im Fall der Facts Bearbeitung erfüllt werden.

6. Deaktivate Effects

Die Deaktivate Effects - Welche die Abfragen an den Reasoner darstellen - können theoretisch beliebig durch Klammerungen und boolische Operatoren verknüpft werden. Es dürfte sich schwierig erweisen ein Interface zu entwerfen, welches keine Kenntnisse der boolischen Logik verlangt. Jedoch kann diese Aufgabe deutlich erleichtert werden, indem man den komplexen Ausdruck in seine Bestandteile, die mit `AND` und `OR` Operatoren verknüpft sind, trennt.

Ein wichtiges Merkmal dabei ist, dass diese boolisch verknüpften Bestandteile im Grunde, auch wenn syntaktisch etwas abweichend, den Formeln, wie wir sie bei den Update Facts gesehen haben entsprechen.

Nochmals an dem Szenario des Beispieldialogs betrachtet:

Es gibt ein Infoblock mit folgenden Facts:

```
ibd4 Wissenschaft beschreibt
ibd4 Weltbild beschreibt
```

Jetzt sollen alle Infoblocks deaktiviert werden, die sich zwar dem Thema `Weltbild` widmen, aber nicht dem Thema `Relion`. Dafür setzt sich die Abfrage aus diesen beiden Elementen zusammen:

```
NOT (X Religion beschreibt) AND (X Weltbild beschreibt)
```

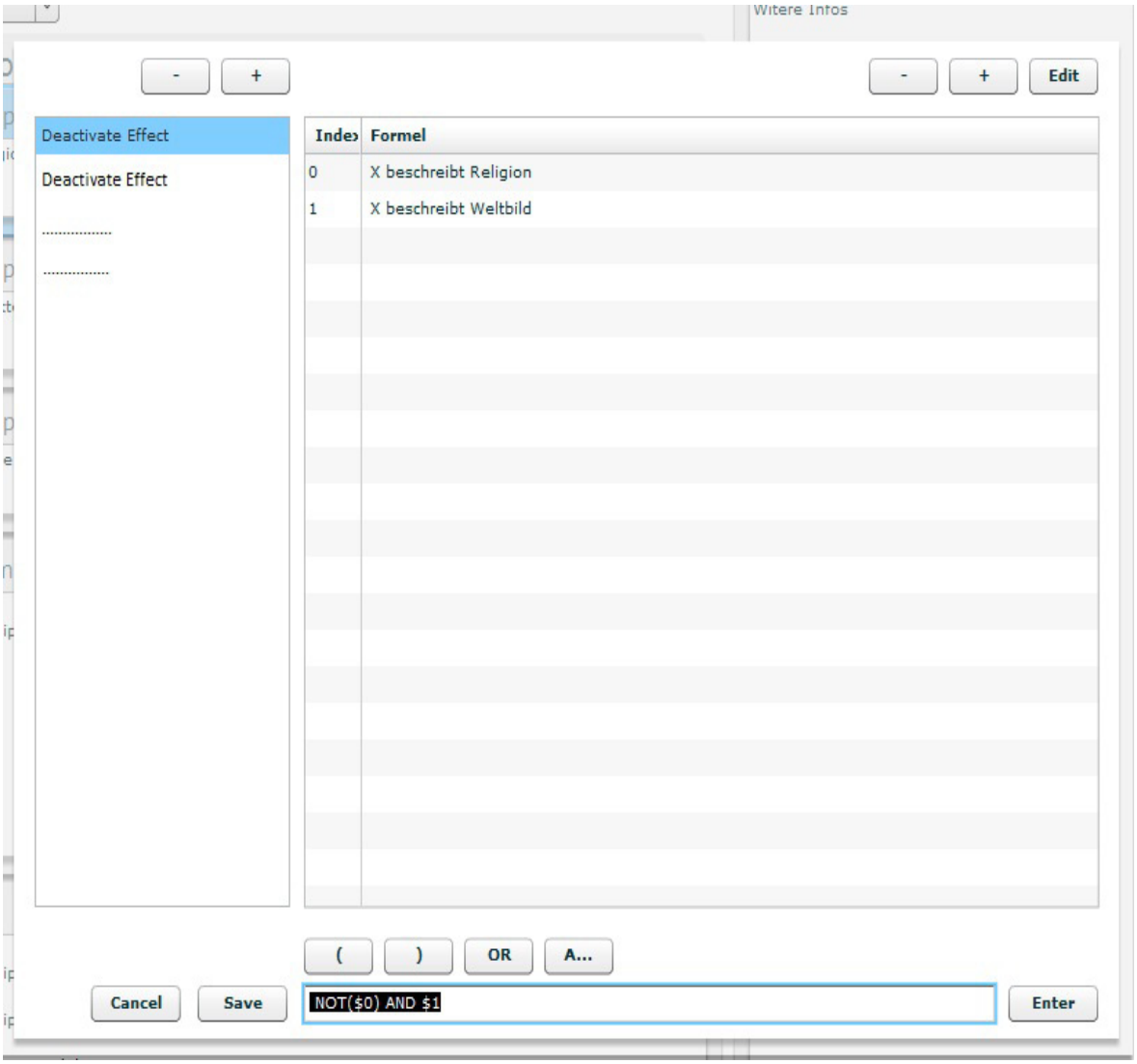


Abbildung 4.4: FormulasEditor

In der [Abbildung 4.4] wird die obige Formel dargestellt. Auf der linken Seite der Maske in der [Abbildung 4.4] sieht man ein Listwidget mit allen Deactivate Effects, die zu einer Responce gehören. Über die + und - Buttons über der Liste können Effects angelegt und gelöscht werden. Mit einem Doppelklick auf ein Element der Liste werden die einzelnen Formelbestandteile in einer textuellen Kurzdarstellung in der Tabelle auf der rechten Seite dargestellt. Für die Eingabe der Teilformeln kann wieder die gleiche Eingebmaske aus der [Abbildung 4.3] verwendet werden, mit dem Unterschied, dass nun die InfoBlock ID durch einen X ersetzt wird. Es wird nur noch ein Formular benötigt um diese Teilausdrücke zu verbinden. Der +, und Edit Button über

der Tabelle öffnet die aus [Abbildung 4.3] bekannte Maske in einem Popup, mit dem ein solcher Teilausdruck eingegeben bzw. bearbeitet werden kann, denn wie man weiß sind die Teilausdrücke semantisch gleich.

Das verknüpfen der Teilausdrücke geschieht in der unteren Eingabezeile, ähnlich wie in dem bekannten Excel Programm. Durch Doppelklicks können die Zeilennummern aus der Tabelle in die untere Leiste übernommen und durch die `AND`, `OR`, "`(` „ und „`)`" Buttons zu boolischen Ausdrücken verknüpft werden.

Idealerweise würde das System eine Vorschlagliste aller bis dato eingegebener Teilformeln der Facts liefern. Wie man jedoch später im Architektur Kapitel sehen wird, ist dieses mit dem gewählten Persistenzmechanismus eine herausfordernde Aufgabe.

7.Richtext
Für diese Anforderung wird das Standart Flex RichText Widget verwendet. Die Anforderung wird nun exemplarisch umgesetzt.

3. Schlusswort

Dieses Kapitel setzte für alle Requirements eine graphische Umsetzung um oder veranschaulichte zumindest einen theoretischen Ansatz. Das Interface Design erhebt keinen Anspruch der Vollständigkeit und besten Usability, soll aber genügend Theoriestoff für eine tatsächliche Umsetzung bieten. Im Folgenden Kapitel wird ein Konzept, die Architektur der Anwendung erläutert.

5. Architektur

5.1.Einleitung

Dieses Kapitel erläutert zunächst grundlegende softwarearchitektonischen Muster und Richlinien für die Implementierung der Editor Rich Internet Applikation und stellt schließlich eine Übersicht der tatsächliche Architektur da. Dabei gliedert sich das Kapitel wie die Webanwendung in zwei Bereiche: Frontendarchitektur und Backendarchitektur.

Unter Frontend versteht sich hier dir tatsächliche graphische Applikation, die auf dem Clientrechner abläuft. Diese sendet Anfragen an den Webserver auf dem das Dialogsystem, welches auf der Persistenzschicht arbeitet und mit dem RACER Reasoner kommuniziert, abgelegt ist. Das Backend ist eine serverseitige Erweiterung der Dialogsystem Applikation, welche dafür zuständig ist die Anfragen des Editors zu beantworten.

Die größere Aufmerksamkeit richtet sich besonders auf den Frontend Bereich, da dieser in der gegenwärtigen Applikation deutlich größer ausfällt. Deswegen, einige Vorgehensweisen analysiert werden um die grobe Richtlinie zu definieren. Für die Backend Architektur, wo ein es Standartverfahren für die Service Implementierung gibt, liegt der Fokus besonders auf der Problematik des Arbeiten mit der Wissenbasis, also der Persistenzschicht und dem Reasoner RACER.

5.2. Frontend Architektur

5.2.1 Einleitung

Das Frontend ist eine autonome Anwendung, welche on demand in dem Flashplayer Plugin des Clientbrowsers abgespielt wird. Hier finden Benutzerinteraktionen statt. Die Interaktionen resultieren in Kommunikation mit bestimmte Services im Backend. Dies Kommunikation wird durch das BlazeDS Framework realisiert.

Ein Standardrezept, eine solche graphische Benutzerschnittstelle zu realisieren ist das bekannte Model View Controller Pattern. Doch dieses Pattern ist heutzutage keine strikte Richtlinie. Es haben sich gemäß den neuen Technologien Abwandlungen gebildet. Gerade das Paradigma, die graphischen Klassen deklarativ definieren zu können und sie mittels DatBinding auf Änderungen des Models reagieren zu lassen, löst eine Diskussion zu Findung eines geeigneten Presentation Patterns in Flex auf.

Motiviert von dieser Problematik wurde gerade in der Flex Community versucht eine Musterlösung zu finden. So sind mehrere sogenannte Application Frameworks entstanden. Einige bringen dabei eine bereits fertige MVC Implementierung mit, andere nennen sich Dependency Injecntion Frameworks und erlauben eine lose Kopplung der Komponenten bei der Umsetzung seines bevorzugten Presentation Patterns.

Die Editor Anwendung ist in ihrem Aufbau an die Adobe Cairngrom 3 Blaupause [AOS10] angelehnt, welche weder eine MVC Implementierung noch ein Dependency Injenction Framework ist, sondern eine Ansammlung der Richtlinien und Tools ist und hauptsächlich auf dem Presentation Model Pattern basiert.

In diesem Kapitel wird die Motivation zu dieser Entscheidung in der Frontendarchitekturleitlinie in Bezug auf konkrete Problematiken der geforderten Anwendung nachvollzogen. Zunächst werden die Nachteile des klassischen MVC Musters für Flex erläutert. Weiterhin wird unter dem Stichwort „MVC Antipattern“ eine Technik, der sich bekannte Flex MVC Implementierungen bedienen, vorgestellt und auf ihre großen Nachteile untersucht. Motiviert von der Beseitigung dieser Nachteile werden weitere Presentation Pattern an einem vereinfachten Szenario aus der Editorapplikation vorgestellt und die Entscheidung zu Gunsten von Presentation Model Pattern nachvollzogen. Schließlich werden zwei Realisierungsmöglichkeiten des Presentation Model Patterns dargestellt und die Vorteile der hierachielosen Implementierung mittels eines Dependency Injection Frameworks dargestellt.

5.2.2. Smalltalk-80 MVC

Seit den 80 Jahren wird, wann immer es darauf ankommt ein graphisches Interface zu erzeugen, zu einer Dreiteilung der Anwendung in Model, View und Controller Komponenten geraten. Dies war eines der ersten Pattern und wurde zusammen mit der Programmiersprache Smalltalk eingeführt. Dies Kapitel erläutert kurz warum dieses Pattern für Flex obsolet ist.

In [LR06, Kap 8.2] findet sich eine Auseinandersetzung mit dem MVC Pattern, wie es ursprünglich gedacht war. Daraus können sofort ohne genaue Erläuterung des Prinzips folgende Nachteile für eine Umsetzung in für die Editor Anwendung in Flex erkannt werden:

View und Controller treten immer als Paar auf und kennen jeweils den anderen, was zu einer engen Kopplung zwischen den beiden führt.

Ein Controller in Smalltalk MVC Verständniss die Aufgabe bestimmte Benutzer Eingaben über die Tastatur oder Maus entgegenzunehmen und diese an das Model weiterzuleiten. Diese Funk-

tionalität ist in Flex bereits in allen graphischen Klassen implementiert und kann über Event-Listener in ActionScript oder InlineEvents in MXML direkt realisiert werden, so dass so eine Art Controller nicht mehr notwendig ist.

Zu Reaktion auf Model Änderungen wird das bekannte Observer Pattern verwendet. Dieses kann durch die Flex DataBinding Funktion vollständig verworfen werden.

Die gesamte Interface Logik bleibt in den graphischen Klassen verkapselt und ist somit schwer ohne ein GUI Automatisierungswerkzeug testbar. (e.g. Der „Speichern" Button wird est sichtbar wenn Änderungen an dem Formular erzeugt wurden). Desweiteren wird dadurch die Code Wiederverwendung in den Views erschwert.

Die Motivation bei der Suche nach einem geeigneten Presentation Pattern geht also viel weiter als ein Werkzeug zur Reaktion auf Benutzereingaben und Synchronisation der Ausgabe zu finden wie es beim klassischen MVC der Fall war und in Flex zu State of the Art gehört. Vielmehr soll die Interface/ Presentationslogik aus den Views ausgelagert werden und eine Möglichkeit gefunden werden wie der Nachrichtenfluss in der gesamten Applikation auf eine lose gekoppelte Weise stattfinden kann.

5.2.3. MVC Antipattern

Im vorherigen Kapitel wurde deutlich, dass das klassische MVC Muster in Flex nicht sinnvoll ist. Dieses Kapitel wird eine weitere unsaubere Vorgehensweise bei der Trennung von Verantworlichkeiten aufzeigen. Eine Vorgehensweise, die sich in vielen bekannte MVC Implementierungsframeworks für Flex wiederfindet, weswegen diese als Standartlösung für die Editoranwendung abgelehnt wurden.

Bei der Suche nach einer Vorlage zu Implementierung des MVC Musters in Flex findet man schnell einige Beispiele wie dieses mit unter Verwendung des Singelton Patterns erreicht werden kann. Dies geschieht folgender Maßen:

Das Model wird über das Singleton Pattern über die Klassendefenition für jeden View und Controller global erreichbar.

```
public class Model
{
        //Singelton Instanz
        static private var _instance:Model;
```

```
            [Bindable] public var data:ArrayCollection;
            public static function getInstance():Model
            {
                  if (_instance == null)
            {
                        _instance = new Model();
            }

          return _instance;
            }
      }
```

Im oberen Beispiel implementiert die Klasse Model das Singelton Pattern in der Methode `getInstance()`. Über den Aufruf von `Model.getInstance()` bekommt jede Klasse nun die gleiche `Model` Instanz, welche beim ersten Aufruf von `getInstance()` im statischen Speicher der `Model` Klasse abgelegt wird. Desweiteren kennzeichnet Model das Property `data` für das Data-Binding.

Der View ruft nun durch InlineEvents bestimmte Funktionen im Controller auf. Der Controller macht Änderungen an dem Model und der View kann sich über DataBinding zum Model aktualisieren.

```
<mx:View>
      <mx:Script>
            <![CDATA[
                  private var controller:Controller = new Controller();
                  private var model:Model = Model.getInstance();
            ]]>
      </mx:Script>

      //DataBinding des List Widgets an das Model data
      <mx:List dataProvider="{model.data}" />

      <mx:Button click="controller.getData()"/>
</mx:View>
```

In der oberen `View` Kasse wird ein neuer `Controller` erzeugt und das `data` Property des Model

Singeltons an das `List` Widget gebunden. Das Button Widget ruft nun über den InlineEvent `click` den EventHandler `getData` im Controller Objekt auf (also beim click auf den Button).

```
public class Controller
	{
		public function getData()
		{
			//Bearbeite das Model
			Model.getInstance().data = new ArrayCollection
			(
				[„Data1“, „Data2“, „...“]
			);
		}
	}
```

Der Controller beschafft sich ebenfalls eine Referenz zu dem Model Singelton über die Klassendefinition und kann es nun manipulieren (hier wird es mit Dummy Daten gefühlt), was eine automatische Aktualisierung des Views zu Folge hat.

Durch die oben erläuterte Vorgehensweise wird durchaus ein Trennung der Verantwortlichkeit erreicht. Vielmehr scheint es zunächst so als könne man leicht ein lose gekoppelte Kommunikation zwischen den einzelnen Komponenten realisieren. Zwei Komponenten können sich über das gleiche Singelton synchronisieren ohne dass diese Referenz über lange Aufrufhierarchien übergeben wird.

Genau dieses Prinzip nehmen sich die bekannten Flex MVC Implemntierungen wie Adobe Cairngrom 2 oder Pure MVC zu Grunde.

Cairngorm geht sogar soweit, dass jede einzelne Funktionalität in eine separate Commandklasse ausgelagert wird. Das Framwork bringt ein zentrales Eventdispatching Mechanismus mit. Dabei implementiert der Entwickler das Auslösen von Customevents über den zentralen EventDispatcher als Resultat auf die Benutzerinteraktionen. Diese werden von einem Frontcontroller verarbeitet wo zu jedem Customevent die `execute` Methode einerr eigen implementierten Commandklasse ausgeführt wird. Jede Commandklasse beschafft sich die Referenz zum Model über das Singelton Pattern. Ebenfalls über das Singelton Pattern kann eine Referenz zu dem Webservice Delegaten beschafft werden über den die Daten vom Server angefordert und im Model gespeichert werden. Entsprechend sind die Views an die Models per DataBinding verknüpft, womit die Vorgehensweise im Kern dem obigen Beispiel entspricht. [vgl. WebLeo08, Part 5]

Würde der Editor also in Cairngorm 2 implementiert werden, würde es bedueten, dass man für jede Benutzeraktion (e.g infoblocks laden, infoblock auswählen) Immer eine eigene Commandklasse anlegen muss. Und diese Klassen beschaffen sich die Referenz zu den Model Daten (Hier zum InfoBlock) immer über das Singelton Pattern.

In [WebLeo08, Part 4] wird diese Methode und die Motivation dahinter genauer unter „Feature Driven Development" erläutert. In einem großen Projekt mit vielen Entwicklern und ständig ändernden und wachsenden Features soll möglichst lose Kopplung erreicht werden, soll heißen die einzelnen Komponenten sollen möglichst unabhängig von einander sein.

Die obige Kurzerläuterung von Cairngorm verdeutlicht die exessive Nutzung des Singelton Patterns in den bekannten Flex MVC Frameworks. Doch obwohl die Anstrebung vom lose gekoppelten Komponenten und getrennter Verantwortlichkeit durchaus erstrebenswert ist, so ist der Weg über das Singelton Pattern der falsche. Warum dies der Fall ist zeigt Misco Hevery auf [Hev10]. Durch so eine starke Verwendung von Singeltons implementiert man ein Äquivalent zu globalen Variablen. Somit ist der gesamte Zustand der Anwendung global. Dass globaler Zustand viele Nachteile hat ist seit langem in der Informatik bekannt. Einige davon sind:

- Verschleierung von Abhängigkeiten bei der Initialisierung (Initialisierung ist über mehrere Instanzen verteilt)

- Veränderung des Ergebnisses durch Ausführungsreihenfolge und Verschleierung der Ausführungsreihenfolge (Seiteneffekte)

- Schlechte Testbarkeit durch Seiteneffekte und Erschwerung des Mockens von Objekten

Die vielen Nachteile des Singelton Patterns haben dazu geführt das es kürzlich als AntiPattern bezeichnet wird. Leider wird es in den bekannten Flex MVC Frameworks durchgehend verwendet. Die Nachteile von globalem Zustand können sich relativieren wenn man strikt die Regeln eines Frameworks befolgt, jedoch gibt es mittlerweile bessere Lösungen, welche wir im weiteren Verlauf betrachten wollen.

5.2.4. Presentation Patterns

In den beiden vorherigen Kapitel wurden einige in der Praxis übliche aber nachteilhafte Techniken zu Realisierung der Clientarchitektur in einer Rich Internet Anwendung dargestellt. Nun wollen wir 2 Verfahren betrachten, die sich in der Praxis als tauglich erwiesen haben und ins Flex

Paradigma passen.

5.2.4.1 Pattern Übersicht

Martin Fowler erläutert in einigen Artikeln auf [Fow10] weitere Presentaion Patterns als Erweiterung zu seinem Buch [Fow02]. Diese Gedanken greift Paul Williams in seinem Artikel [Will07] auf und erläutert die bedeutenden Pattern anhand von beispielhaften Flex Rich Client Implementierungen. Folgende Muster werden analysiert:

- Autonomous View
- Supervising Presenter / Controller
- Presentation Model
- View Helper
- Code Behind
- Passive View

Für die Realisierung des Flex Clients kommen für mich 2 Muster in Frage: Supervising Presenter und Presentation Model.

Die Nachteile der anderen Pattern können aus dem Artikel [Will07] entnommen werden:

Autonomous View ist nicht wirklich ein Pattern, sondern die Art und Weise ganz auf eine Architekturteilung zu verzichten und die Anwendung allein aus MXML Modulen mit Scriptblöcken aufzubauen. View Helper wird von dem Author lediglich aufgeführt um seine Nachteile für Flex aufzuzeigen. Code Behind stammt aus .NET / WPF Welt. Die Implementierung dieses Patterns in Flex ist nicht eindeutig, führt zwar zu Auslagerung des Presentationslogik Codes in separate ActionScript Files, bringt aber keine signifikante Verbesserung im Trennen von Verantwortlichkeit. Und schließlich Passive View ist zwar gut auf dem letzen Gebiet, beraubt den Entwickler aber einiger mächtigen Funktionalitäten der MXML Sprache.

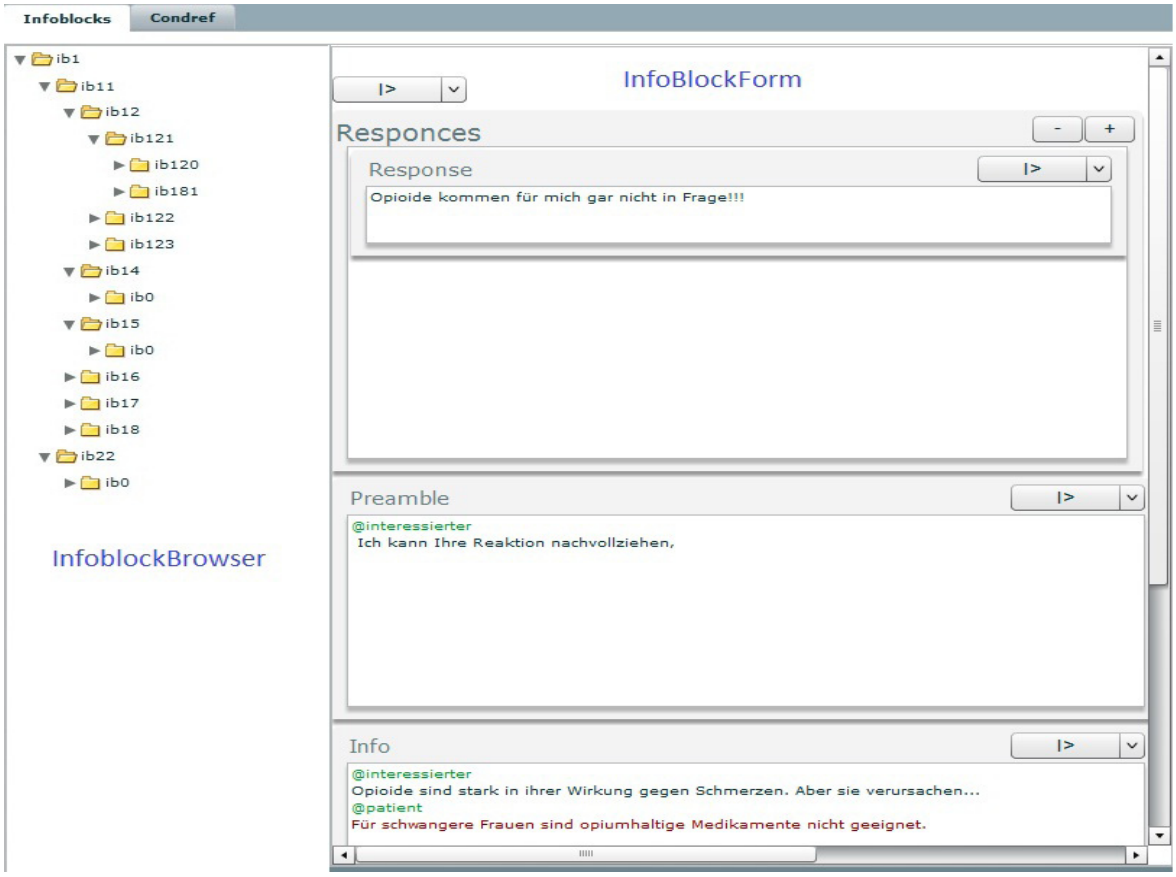
Um die Pattern zu veranschaulichen hat Paul Williams für jedes eine kleine Beispielapplikation umgesetzt. Dieses Szenario findet sich ebenfalls in der Editor Appliaktion wieder, so dass die beiden restliche Pattern anhand dessen unter geringer Vereinfachung betrachtet werden können.:

Ein Benutzer Soll aus einer Liste von Infoblocks Íd's auswählen können und nach der Auswahl soll in das gewünschte Infoblock mit seinen Aufbauelementen (Responces, Preambel, Info, Challenge) angezeigt werden. [Abblildung 5.1]

Dabei soll ein simples UseCase in beiden Pattern Paradigmen betrachtet werden. Nach dem der

Benutzer eine Liste der Infoblocks vom Server erhalten hat, selektiert er in der Liste (Infoblock

Abbildung 5.1: Presentation Patterns Szenario



BrowserView) eine bestimmte InfoBlock ID und die Informationen dieses Infoblocks werden auf der rechten Seite angezeigt (InfoblockFormView). Vor dem Anzeigen eines Albums wird geprüft ob der Benutzer zuvor Änderungen an dem vorher angezeigtem Album durchgeführt hat, die nicht gespeichert wurden und falls nicht wird das neue Album angezeigt, ansonsten nicht. Genau diese Inteface Logik wird in beiden Pattern in die Presenter bzw. Presentation Model Klasse ausgelagert.

5.2.4.2. Supervising Presenter

Schlüsseleigenschaften [vgl. Will07, Supervising Presenter] :

- Zustand (Model) ist in den Views
- Logik ist in dem Presenter
- Presenter beobachtet den View
- Presenter aktualisiert den View

- Presenter kennt den View, View kennt den Presenter nicht.

Der Presenter trägt sich als EventListener zu den einzelnen Widgets der View Klasse. Wenn eine Benutzerinteraktion stattfindet, so werden entsprechend eingetragene Eventhandler in der Presenterklasse aufgerufen. Die Handler schicken über die Delegates Anfragen an den Server, führen die Interface Logik aus und triggern schließlich Aktualisierungen in dem View (direkt oder per Modelmanipulation auf welche der View mittels Databinding reagiert).

Betrachten wir das Ganze an dem anschaulichen Beispiel:
Die Applikation besitzt 2 View Klassen - `InfoBlockBrowser`, `InfoblockForm` und entsprechend 2 Presenter. Dabei kennt `InfoBlockBrowser` die `InfoblockForm` (automatisch durch MXML Schachtellung) und eine entsprechende Beziehung besteht zwischen den Presentern (wird manuell aufgebaut).[Abbildung 5.3]

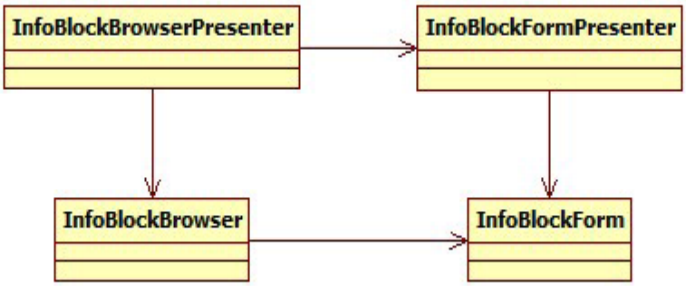


Abbildung 5.2: Supervising Presenter Klassendiagramm

Folgende Schritte laufen bei dem definiertem Usecase ab [Abbildung 5.4]:

Vorbedingung:

1. Der `InfoblockBrowserPresenter` trägt den Eventhandler `changeSelectedInfoBlock()` zu dem `ItemSelected` Event des List-Widgets in der InfoblockBrowser Klasse. Bei Auswahl einer ID im Browser wird also diese Methode des Presenters aufgerufen.

Ablauf:

1. Der `InfoblockBrowserPresenter` fordert die Liste aller InfoBlocks von dem Server über den Delegaten an und legt diese in der Membervarible `infoblocks` im InfoblockBrowser, welches seine Anzeige wiederum durch DataBinding aktualisiert.
2. Der Delegate liefert die Liste aller InfoBlocks zurück.
3. Der `InfoblockBrowserPresenter` überträgt die Liste der InfoBlock in den Browser, worauf dieser seine Anzeige aktualisiert

4. Der Benutzer clickt mit der Maus eine Infoblock ID in der Liste an, worauf der `changeSelectedInfoBlock()` Handler im `InfoblockBrowserPresenter` aufgerufen wird. Dieser entscheidet ob jetzt der neue InfoBlock sofort angezeigt wird, weil es keine ungespeicherten Änderungen am vorher ausgewählten InfoBlock gibt, oder zuvor ein Bestätigungsdialog (Ja/Nein) kommt, falls vergessen wurde vorher zu speichern und delegiert bei positiver Prüfung den Aufruf an den `InfoBlockFormPresenter` mit dem selektierten Infoblock.
5. Der selektierte InfoBlock wird nun wiederum schleißlich in die `InfoBlockForm` übertragen, worauf diese die Anzeige aktualisiert.

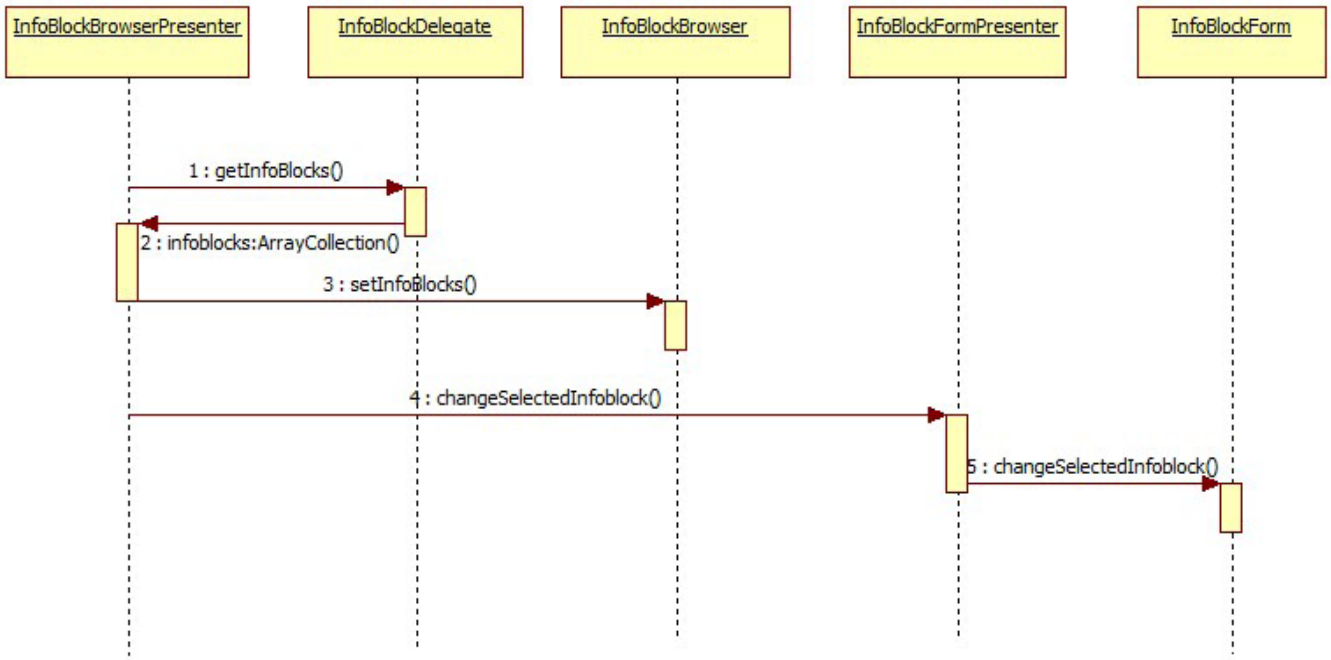


Abbildung 5.3: Supervising Presenter Sequenzdiagramm

PseudeCode des InfoBlockBrowser Presenters:

```
public class InfoBlockBrowserPresenter {
   // Corresponding view
   private var _infoBlockBrowser : InfoBlockBrowser;
   // Sub-presenters
   private var _infoBlockFormPresenter : InfoBlockFormPresenter;

   //constructor
   public function InfoBlockBrowserPresenter{

       //infoblocks vom Server Anfordern und den View aktualisieren
```

```
        var delegate : InfoBlockDelegate = new InfoBlockDelegate();
        _infoBlockBrowserView.infoblocks = delegate.getInfoBlocks();

        //trage einen Eventlistener zu dem InfoblockView List Widget ein
        _infoBlockBrowserView.list.addEventListener(
           ListEvent.CHANGE, changeSelectedInfoBlock
        );

    }

    //EventHandler
    public function changeSelectedInfoBlock{
       if (_infoBlockFormPresenter.infoBlockChanged()){
          //....
                //Zeige Bestätigunsmeldung (Wollen sie zuvor speichern? JA/NEIN)
       } else { selectInfoBlock(); }
    }

    public function selectInfoBlock(){
       _infoBlockFormPresenter.changeSelectedInfoBlock(
              _infoBlockBrowser.list.selectedItem
        );
    }
}
```

5.2.4.3. Presentation Model

Schlüsseleigenschaften [vgl. Will07, Presentation Model]:

- Zustand und Logik sind im Presentation Model (PM)
- Der View beobachtet das PM und aktualisiert sich selbst
- Der View kennt das PM, das PM kennt jedoch den View nicht

Das Presentation Model ist hier eine abstrakte Abbildung des Views. In Anderen Worten gibt das PM den Zustand und das Verhalten der graphischen Klassen wieder ohne sich mit der graphischen Darstellung auseinander zu setzen. Der View ist also eine Projektion oder Wiedergabe der Presentation Models auf dem Bildschirm.
Das muss man sich folgender Maßen vorstellen. Angenommen es existiert ein Bereich auf der

Benutzeroberfläche, der erst sichtbar wird, wenn ein Benutzer ein bestimmtes Widget angeklickt hat. Z.B. der „Absenden“ Button wird erst sichtbar, wenn per Checkbox die AGB's akzeptiert wurden. So wäre der normale Weg diese Logik zu implementieren, ein EventListener zu dem Checkbox Widget, der direkt in der View Klasse das „visible“ Property des „Speichern“ Buttons manipuliert.

Im Verständnis des Presentation Model Patterns, wird diese Logik in die PM Klasse ausgelagert. Diese besitzt eine boolische Instanzvariable „agb_accepted“. Per Databinding wird das „visible“ Property des Speichern Buttons an diese Instanzvariable in der PM Klasse verknüpft. Ändert man jetzt den Zustand der „agb_accepted“ variable (etwa durch ein Listener zu dem Checkbox-widget), so aktualisiert sich der View automatisch und blendet den Button ein bzw. aus. Natürlich sind auch komplexere Aktualisierungmechanismen denkbar.

Betrachten wir das ganze an dem obigen Beispiel. Wieder gibts 2 View und entsprechend 2 PM Klasse. [Abbildung 5.4]

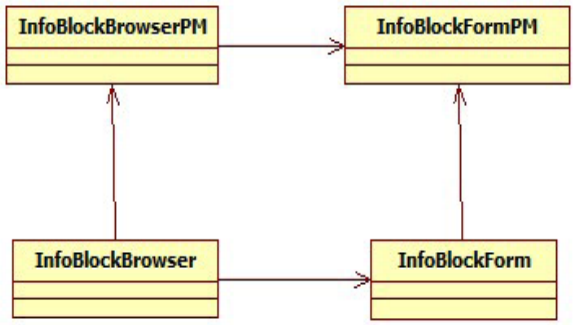


Abbildung 5.4: Presentation Model Klassendiagramm

Der Anwendungsfall läuft nur folgender Maßen ab:

1. Der `InfoBlockBrowser` ruft die getInfoBlocks() Methode des `InfoBlockBrowserPM's` auf.
2. Das `InfoBlockBrowserPM` fordert die Liste aller Alben durch den Delegate vom Server an.
3. Der Delegate liefert die Liste aller InfoBlocks zurück und das `InfoBlockBrowserPM` speichert die Liste in der eigenen Instanzvariable infoBlocks.
4. Der `InfoBlockBrowser` aktualisiert die graphische Liste der `InfoBlocks` durch DataBinding zu `InfoBlockBrowserPM.infoBlocks`
5. Der Benutzer clickt mit der Maus eine InfoBlock ID in der Liste an, worauf der `changeSelectedInfoBlock()` Handler im `InfoblockBrowserPM` aufgerufen wird. Dieser entscheidet ob jetzt der neue InfoBlock sofort angezeigt wird, weil es keine ungespeicherten Änderungen am vorher ausgewählten InfoBlock gibt, oder zuvor ein Bestätigungsdialog (Ja/Nein)

kommt, falls vergessen wurde vorher zu speichern und delegiert bei positiver Prüfung den Aufruf an den `InfoBlockFormPM` mit dem selektierten Infoblock.

6. `InfoBlockFormPM` speichert den neu selektierten Infoblock in seiner Instanzvariable InfoBlock ab.

Die `InfoBlockForm` welche durch das Datbinding die Instanzvariable `InfoBlockFormPM.infoBlock` beobachtet aktualisiert die Anzeige.

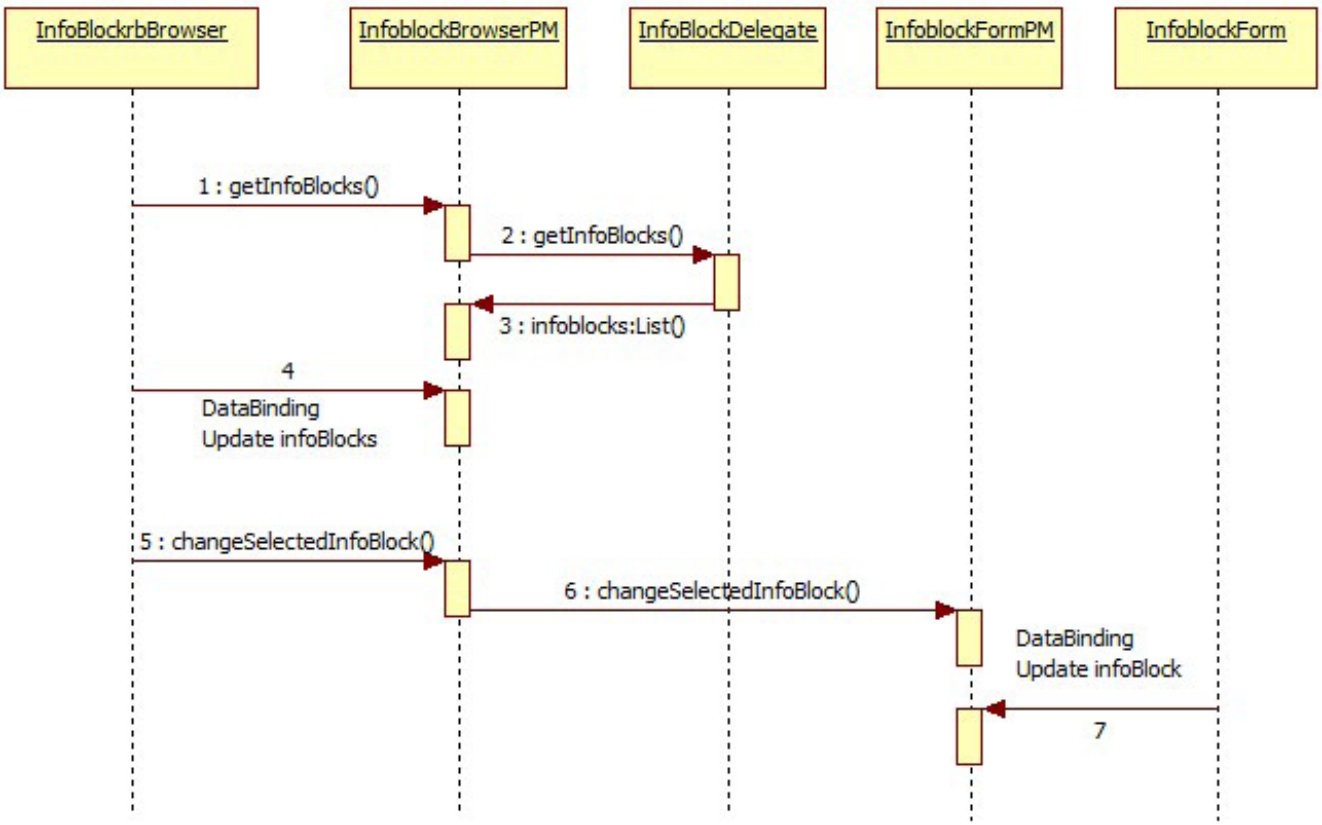


Abbildung 5.5: Presentation Model Sequenzdiagramm

PseudeCode des InfoBlockBrowserPM's:

```
public class InfoBlockBrowserPM {

//Bindable Variable, zu View Aktualisierung
[Bindable]
public var infoBlocks:ArrayCollection;
// Sub-PM
   private var _infoBlockFormPM : InfoBlockFormPM;
```

```
    //Konstructor
    public function InfoBlockBrowserPM{
        //infoblocks vom Server Anfordern und den View aktualisieren
        var delegate : InfoBlockDelegate = new InfoBlockDelegate();
        this.infoBlocks = delegate.getInfoBlocks();

    }

    //EventHandler
    public function changeSelectedInfoBlock(selectedInfoBlock:InfoBlock){
       if (_infoBlockFormPresenter.infoBlockChanged()){
          //....
                //Zeige Bestätigunsmeldung (Wollen sie zuvor speichern? JA/NEIN)
       } else { selectInfoBlock(selectedInfoBlock); }
    }

    public function selectInfoBlock(selectedInfoBlock:InfoBlock){
       _infoBlockFormPM.changeSelectedInfoBlock(
               selectedInfoBlock
        );
    }
}
```

5.2.5 Pattern Analyse

Nach dem zwei Pattern: Supervising Presenter und Presentation Model für die Kernarchitektur der Editoranwendung in Frage gekommen sind und anhand eines Beispiels erläutert wurden sollen diese nun auf ihre Eignung analysiert werden.

Beide Pattern erlauben eine Erzeugung einer separaten Klassenhierachie zur Auslagerung der Presentationslogik. Der Hauptunterschied zwischen den beiden Pattern ist die Beziehungsrichtung zwischen den graphischen Komponenten und den Presenterkomponenten. Daraus ergeben sich die Stärken und Schwächen der beiden Ansätze.

Das Supervising Presenter Pattern erlaubt eine größere Auslagerung der Logik in die Presenter Klasse, weil durch die Referenzierung der graphischen Komponente im Presenter alle graphischen Funktionen direkt angesprochen werden können. Beim Presentation Model hingegen, wo umgekehrt die Presenterklasse von dem View referenziert wird und Updates per Änderung

des Zustands oder Eventauslösung in der Presenteklasse getriggert werden, muss komplexe Anzeigelogik in dem View bleiben. Ebenfalls durch die direkte Referenzierung des Views ist das Pattern viel intuitiver bei der Umsetzung.

Während das Presentaion Model weniger intuitiv ist, so folgt es mehr dem deklarativem Paradigma des Flex Frameworks. Durch InlineEvents (`button.onClick="model.doLogik„`), können Methoden in dem PresentaionModel bequem ausgelöst werden und das DataBinding, ein Kernfeature von Flex wird zu dem Hauptwerkzeug. Ein bedeutend größerer Vorteil ergibt sich bezüglich der Testbarkeit. Der nicht ausgelagerte graphische Code ist ohnehin ohne ein GUI Automatisierungswerkzeug sehr schwer testbar, was aber kein großes Manko ist, weil er ja hauptsächlich aus Aufrufen der Frameworkfunktionalität besteht. Die Applikationslogik hingegen kann aber durch einen simplen Unittest der PM Klasse abgedeckt werden, da eben gar keine Referenzen zu den Views in den PM's vorhanden sind. Beim Supervising Presenter müssen bei einem solchen Unittest die View Objekte gemockt werden, was zwar eine bekannte aber keine triviale Technik ist. Aus diesen Gründen wird für die Umsetzung das Presentation Model Pattern gewählt.

5.2.6. Anwendung des Presentation Patterns

In dem vorherigen Kapitel wurden 2 Presentation Patttern dargestellt: Supervising Presenter und Presentation Model und nach einer Analyse wurde die Entscheidung zu Presenetation Model gefällt. Für diese Analyse wurde ein vereinfachtes Beispiel aus der Editorapplikation herangezogen. In diesem Kapitel wollen wir nun betrachten welche Problematiken auftauchen, falls man versucht die Editorapplikation ähnlich dem Beispiel aus dem vorherigen Kapitel zu Implementieren. Schließlich wird erläutert wie mithilfe eines Dependency Injection Frameworks diese Problematiken entschärft werden können.

5.2.6.1. Hierachischer Ansatz

Unbetrachtet dessen ob man sich für Supervising Presenter oder Presentation Model Pattern als Basis für die Client Architektur entscheidet, wird das Problem der lose gekoppleten Kommunikation nicht gelöst. Mit dieser Problematik setzt sich Tom Sugden auseinander [Sug09].

In dem Beispiel aus 5.2.4. Presentatio Patterns spiegeln die Referenzen zwischen den einzelnen Presentation Model Klassen exakt die Referenzen zwischen den Views wieder. Diese Vorgehensweise nennt Sugden „hierachischer Ansatz".

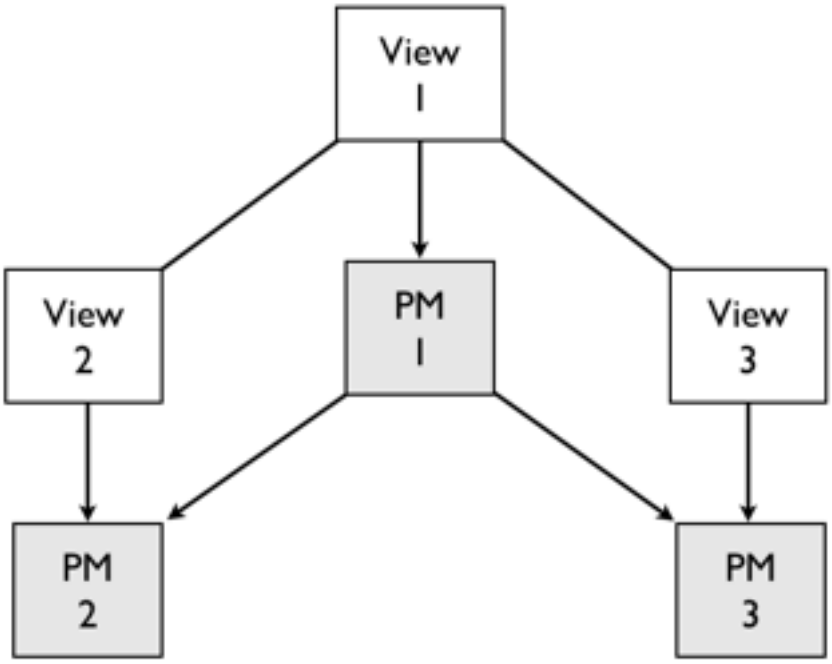


Abbildung 5.6: PM Hierachischer Ansatz [Sug09]

Bei diesem Ansatz wird das oberste Presentation Model mit allen Kindern in dem Top View isntaziert und Kinder PM's werden in der View Hierarchie heruntergereicht.

```
<example:MyComponent ... >
<mx:Script>
       //Model wird beim Aufruf von MyComponent gesetzt
      [Bindable]
      public var model:MyComponentPM;
  </mx:Script>
 //Rufe MyChildcoponent auf, setzte dessen PM über die Referenz zum eigenen PM
  <example:MyChildComponent model="{ model.myChildComponentPM }" ... />
</example:MyComponent/>
```

Wir sehen hier die Definition einer MXML Komponente `MyComponent`, also eine View Klasse mit dem `[Bindable]` Property `model`. `MyComponent` besteht wiederum aus `MyChildComponent` welches ebenfalls ein `[Bindable]` Property `model` hat. Keines dieser Komponenten instanziert sein PM direkt. Die PM Hierachie wird von Außen erzeugt und beim Aufruf wird durch das XML Attribut `model` und die geschweiften Klammern Syntax das Binding der Kinderkomponente zum Kindes PM über die Referenz zum eigenen PM gesetzt- `MyChildComponent model="{ model.`

`myChildComponentPM }" .`

„Wenn eine solche Hierachie erzeugt wurde, findet die Koordination durch Methodenaufrufe zwischen den PM's, teilen von Objekten und abfeuern von Events die Hirarchie hoch statt", so Sugden. *„Wenn also der Benutzer ein Item aus einer Liste in View 2 selektiert, wird ein selectedItem Property in PM 2 gesetzt und es wird ein Event abgefeuert um diese Auswahl bekannt zu geben. PM 1 könnte auf dieses Event hören und seine eigene Logik als Antwort darauf ausführen",* schreibt Sugden weiter.

Das Ganze Gebilde wird aber weit aus komplexer und unangenehmer in der Implementierung, wenn man weitere Szenarien überlegt. Die obere Pseudocode Anleitung zu Aufbau der View und PM Hierachie aus dem Artikel von Sugden, setzt voraus, dass die View Hierarchie starr ist. Die View Komponenten sind quasi schon von vornerein da (auch wenn nicht eingeblendet) und warten nur darauf mit Daten gefüllt zu werden.

An dieser Stelle betrachten wir kurz die Infoblockdarstellung [Abbildung 4.2]. Der Infoblock enthält bekanntlich eine variable Anzahl von Responces und die Responces eine variable Anzahl an Effects. Es liegt die Schlussfolgerung nahe, dass für jedes dieser Elemente eine eigene View Klasse(MXML Custom Component) und nach dem Presentation Model Pattern entsprechend eine PM Klasse pro View Komponente gebildet werden sollte. Es sollen also ein InfoBlock View welcher eine Liste von Responce Views kennt und analog dazu ein InfoBlockPM, der ein Liste von ResponcesPM kennt erzeugt werden. Die ResponceViews und PM'd kennen wiederum eine Liste von Effect Views und PM's. Man weiß also erst, wenn die Daten vom Server geladen wurden, wie viele ResponceViews zu einem InfoblockView oder wie viele EffectViews zu einer Responce erzeugt werden müssen.

Der PM Graph kann durch in Kauf nehmen einer zusätzlichen Iteration über die Responces des vom Server erhaltenen Infoblocks und der Verkapselung jeder Responce zu einer ResponcePM Klasse erzeugt werden. Wenn die Applikation aber, wie in diesem Fall alle CRUD Operationen nicht sofort an den Server schickt und sich unmittelbar darauf aktualisiert, wie es in normalen Webanwendungen üblich ist, sondern zunächst clientseitig einen Zustand pflegt (siehe Architektur Umsetzung), so entsteht hier ebenfalls zusätzlicher Aufwand beim Löschen und Erzeugen der Elemente, da die PM Listen und die Elementlisten separat gepflegt werden müssen. (also z.B. sowohl die Responces selbst als auch die ResponcePM's)

Viel unschöner wird es aber im Punkt der Kommunikation. Wenn man z.B. ein Element wie Responce anklickt, soll eine kurze Information dazu auf der rechten Seite des Bildschirms in einer Details Textmaske angezeigt werden. Das `DetailsPM` ist also in einem ganz anderen Zweig des Beziehungsgraphen [Abbildung 5.7]. Laut Vorschlag vom Sugden sollte das ResponcePM ein Event abfeuern. Als Listener zu diesem Event kann sich nur das darüberstehende PM eintragen,

also `ResponcesPM`, weil nur dieses die Referenz zu dem abfeuernden Element besitzt. In unserem Beispiel aber enthält eine Reponce wiederum mehrere Effects. Theoretisch könnten Effects ebenfalls weitere Elemente erhalten. Um die Information eines solchen tief liegenden Elements anzuzeigen müsste man bei dem hierachischen Ansatz eine ganze Event Kette auslösen, bei der die Komponenten den Weg nach oben nach und nach auf Events reagieren und neue Events auslösen bis schließlich ein PM an der Reihe ist, der das `DetailsPM` in der Hierachie nach unten referenzieren kann. Dieses Verfahren ist nicht nur unangenehm in der Implementierung, sonder auch durch Erzeugung vieler unnötiger Eventlistener unperformant.

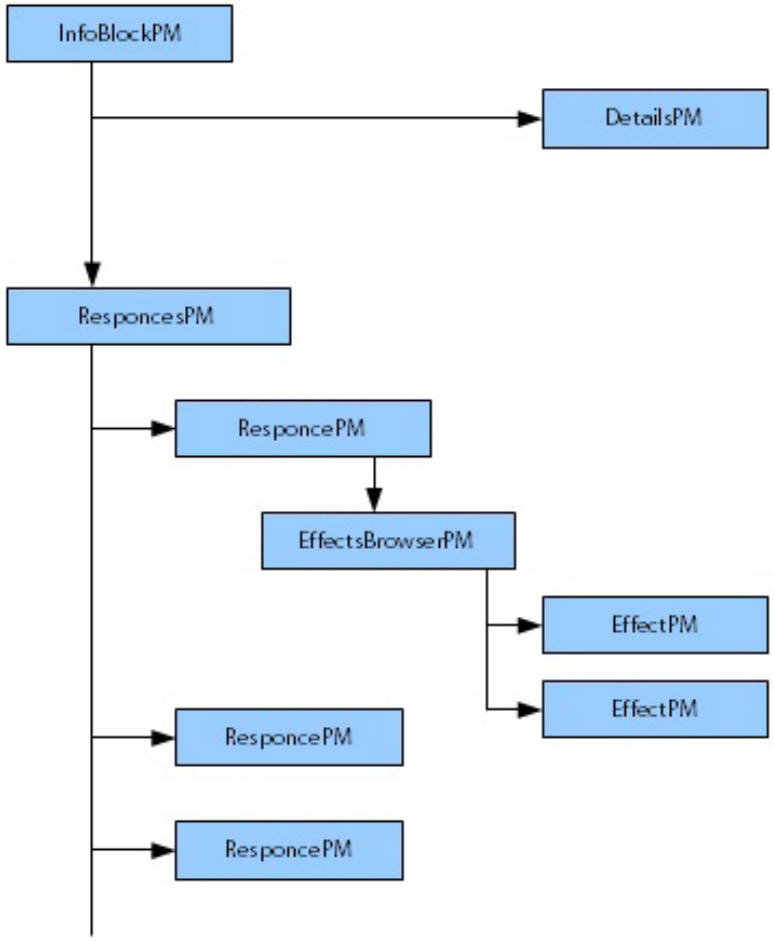


Abbildung 5.7: PM Objektbeziehungen

Abhilfe kann hier das Event Propagation Feature von Flex schaffen[Flex08, Kap4]. Dabei wird eben genau so ein EventBubbling entlang der Komponenten ermöglicht ohne überall Eventlis-

tener und Händler zu deklarieren. Jedoch kann so ein „bubbling“ Event nur entlang der Komponenten, die sich in der sog. Display List befinden - also den Views entlang bewegen. Dieser Mechanismus würde sich also dem Test entziehen. Ein weiterer Lösungsansatz wäre das bekannte Mediator Pattern, hätte aber den Nachteil, dass ebenfalls der Mediator entlang der Hierachieen gereicht werden müsste.

Der Fazit - es ist nicht unmöglich den „hierachischen Ansatz“ mit Flex Hausmitteln zu realisieren, jedoch ist die Gefahr groß in einem Chaos aus Beziehungen zwischen den Objekten zu enden. Außerdem, so führt auch Sugden an, wird das Architekturgebilde ziemlich änderungsunfreundlich, da Änderungen im Design, welche gerade in der Anfangsphase nicht selten sind, sofort Änderungen in dem Referenzenbaum der PM Klassen auslösen.

5.2.6.2. Hierachiloser Ansatz

Der hierachische Ansatz ist mit einigen Problematiken verbunden. Jedoch wurde im Artikel von Sugden bei der Umsetzung diese Anstatzes eine bestimmte Technik, die einen hohen Stellenwert in der Informatik besitzt angewendet. An dem Pseudocode sieht man, dass eine View Komponente sein PM nicht direkt erzeugt, sondern die Referenz von außen eingefügt wird. Dieses erlaubt eine Trennung des Referzengraphen vom Objektinstanziierungsgrpahen. Das Verfahren nennt sich Dependecy Injection (DI). So zeigt auch Marco Hevery [Hev08]diese Lösung als Antwort auf die Singelton Problematik aus [5.2.3. MVC Antipattern].

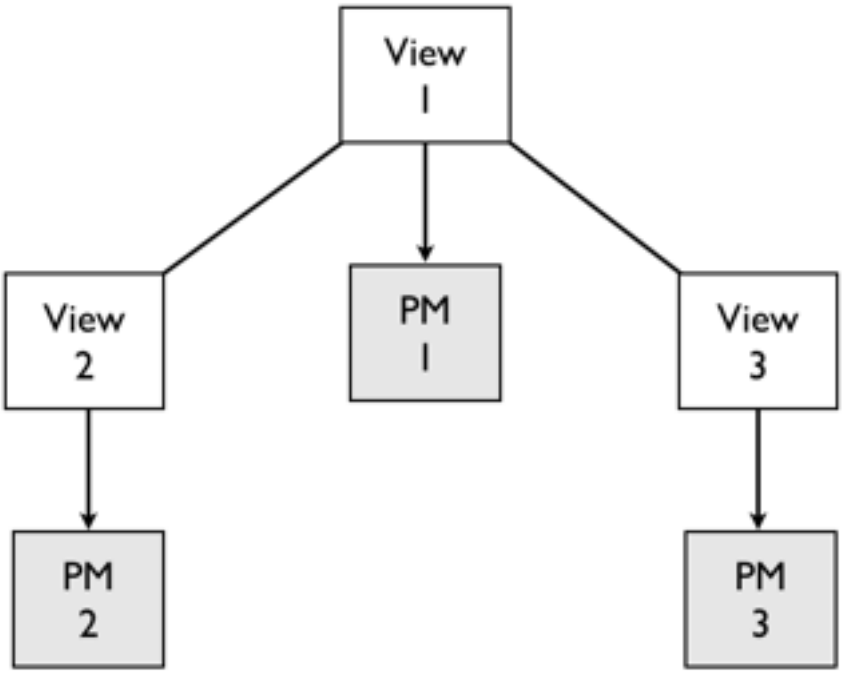


Abbildung 5.8: PM Hierachiloser Ansatz [Sug09]

Es ist jedoch, wie man an dem hierachischen Ansatz gesehen hat, äußerst schwierig, eine Dependency Injection über die gesamte Applikation hinweg zu realisieren. Aus diesem Grund existieren beinahe für jede Programmiersprache Frameworks, die diese Aufgabe erledigen. Oft bringen diese sogar eine Lösung für eine lose gekoppelte Kommunikation in der Anwendung mit.

Beim hierachielosen Ansatz bedient sich Sugden eine solchen Frameworks namens Parsley. Dabei wird zwar eine Hierachie bei den Views, jedoch nicht zwischen dem PM's erzeugt.[Abbildung 5.8] [6]

Betrachten wir das ganze an wieder an einem Pseudocode :

```
<example:MyComponent ... >

   <mx:Script>

      [Inject]
      [Bindable]
      public var model:MyComponentPM;

   </mx:Script>

   <example:MyChildComponent/>

   ...

</example:MyComponent/>
```

Man sieht nochmals eine Komponente mit einer Unterkomponente, jeweils mit dem Property `model`, welches das PM dieser Komponente darstellt . Dieses Mal wird das PM jedoch nicht händisch injected, sondern durch das Framework, was durch das `Inject` Metatag ausgelöst wird. Die Instanziierung der PM Klasse geschieht in einem sog. Kontext des DI Containers.

Will man nun die Presentation Models miteinader koordinieren, so nennt Sugden 3 Möglichkeiten:

6 Die Hierachie zwischen den Views wird per default erstellt. Ein graphisches Element in Flex befindet sich immer in einer sog, Displaylist,welche genauer gesagt ein Baum ist. Dabei hat ein Element mindestens ein Vaterknoten - die Stage. Die View Hierachie wird also automatisch durch das Layout vorgegeben.

- Messaging
  Dieses ist ein Mechanismus, welches die DI Frameworks als Antwort auf die Problematik mitbringen. Über ein zentralen Eventmechanismus des DI Containers, können alle Komponenten welche vom Container bekannt sind, Nachrichten unter einander austauschen ohne sich gegenseitig zu kennen. Diese Kommunikation wird ebenfalls durch Metatags konfiguriert. Die Appliaktion feuert ein ganz normales ActionScript Event ab und der Container sorgt dafür, dass entsprechend der Metadaten Konfiguration der richtige EventHandler in einem anderen vom Container gemanageten Objekt aufgerufen wir.

- Shared Objects
  Unter diesem Stichwort versteht man die Vorgehensweise eine gemeinsame Objekt Instanz in zwei Verschiedene PM's einzufügen. Wenn zwei PM's ein gleiches Objekt teilen, können sie sich unter einander koordenieren, indem sie seine Methoden aufrufen und auf Änderungen dessen Zustandes reagieren

- Presenter
  Dieses ist im Grunde eine Umkehrung der vorherigen Variante, wobei eine Klasse eingeführt wird, in die mehre PM's injected werden. Presenter Methoden werden direkt aufgerufen und dieser koordiniert die Aufrufe an zuständige PM's. Diese Technologie entspricht im Grunde dem Mediator Pattern.

Gerade für den Anwendungsfall des Infoblocks mit mehreren Responces und Effects, wobei mehre Views dynamisch erzeugt werden, ist es ein unentbehrlicher Gewinn auf den Aufbau des PM Referenzenbaums verzichten zu können, da nun lediglich die Infoblock Daten verwaltet werden und nicht mehr die dazugehörigen PM Referenzen. Eine genaue Auseinadersetzung mit diesem Anwendungsfall findet sich in [6.2] und [6.3].

5.2.6.3. Fazit

Der Abschnitt Umsetzung des Presentation Patterns verdeutlichte, dass die Nachbildung der View Referenzen in den Presentation Klassen (PM's) gerade by dynamisch, je nach Datenmenge, erzeugten Komponenten sehr aufwändig ist und zeigte, dass sog. Inversion of Control (IoC) Frameworks eine große Erleichterung bei der Koordination der PM's sind, weil sie eine lose Kopplung im System erlauben.

Das Risiko beim Einsatz eines IoC Frameworks ist minimal, da dieses lediglich wie ein Kleber zwischen den einzelnen Komponenten fungiert und diese nicht abhängig vom Framework macht.

Dessen ungeachtet möchte man sich natürlich nicht auf eine unsichere Technologie verlassen, die noch im Entwicklerstadium steckt. Eine umfangreiche Analyse der IoC Frameworks würde aber den Rahmen dieser Arbeit sprengen. Es gibt aber einige Kriterien die auf den ersten Blick ein gewisses Vertrauen in Parsley wecken.

Zunächst wird Parsley von Powerflasher GmbH supported, welche eine der bekanntesten Flex IDE's FDT entwickelt hat. Das Team hat bereits mehrere RIA's mit Parsley realisiert. Eine weiteres Kriterium ist die Anerkennung von Adobe durch die neue Cairgrom 3 Richtlinie.[AOS10]

Im [5.2.3. MVC Antipattern] wurden die MVC Microarchitekturen wie Adobe Cairngorm 2 angesprochen. Dieses war bis jetzt das Rezept, welches Adobe im Rahmen des Supports bei der RIA Entwicklung in Flex auslieferte. Das nächste Cairngorm wird jedoch keine MVC Implementierung sein, sondern eine Blaupause für die Anwendungsarchitektur. Und diese Blaupause spricht sich ebenfalls für das Presentation Model Pattern in Verbindung mit einem IoC Framework. Dass Einiges dafür spricht, wurde in diesem Kapitel deutlich. Adobe möchte sich nicht für ein bestimmtes IoC Framework festlegen, die Code Beispiele verwenden jedoch Parsley, so dass der Einstieg schnell ist. So wird auch unter anderem der genannte Artikel von Sugden von Adobe Cairngorm referenziert.

Die Adobe Blaupause umfasst viel mehr als das Presentation Model Pattern und die Verwendung von IoC. Auf eine volle Darstellung muss hier leider ebenfals verzichtet werden. Als Referenz is lediglcih [AOS10] zu nennen. Einige Ansätze (wie die Trennung jedes Serverzugriffs in eine eigene Kommandklasse) sind jedoch für die Größe dieser Anwendung überproportioniert oder kommen gar nicht zum tragen.

Als eine Anmerkung zum Schluss möchte ebenfalls hinzufügen, dass eine strikte Einhaltung des hierachilosen Ansatzes, bei dem jede Custom View Komponente immer eine eigene PM Klasse für die Interface Logik besitzt eher theoretisch ist. Denn manche Komponenten besitzen so wenig Interface Logik, dass sie entweder direkt auf den Domain (VO) Objekten arbeiten oder einfach Funktionalität anderer PM's aufrufen könnten.

5.2.7. Architekturumsetzung

Nach dem der theoretischer Analyse zu den besten Praktiken für die Realisierung der clientseitigen Architektur einer RIA in Flex, welche sich für die hierachilose Anwendung des Presentation Model Patterns mittels eines Dependecy Injection Frameworks ausgesprochen hat, sollen die Ergebnisse dieser Analyse auf die Editoranwendung übertragen werden. Von den tiefgehenden Implementierungsdetails soll vorest abgesehen werden. Das Ziel dieses Abschnittes ist den Klas-

sengerüst zu Umsetzung des im Design Kapitel definierten Client Interfaces zu beschreiben. Die Frontendarchitektur teilt sich in 4 Schichten auf:

1. Presentation
2. Application
3. Domain
4. Infrastructure

Die Schichten der Client Architektur werden nun mit ihren wichtigsten Kolloboratueren vorgestellt:

1.Presentation
Der Name dieser Schicht spricht für sich. Hier befinden sich die Klassen, welche die graphischen Masken aus dem Design Kapitel erzeugen. Nach dem Presenetation Model Pattern besitzt jede dieser Maskenklassen ein Presentation Model (PM), welches sein logisches Abbild darstellt und somit ebenfalls in der Presentation Shicht angesiedelt wird.

Die Klassenbeziehungen der View Klassen ergeben sich automatisch aus ihrer Anordnung auf dem Bildschirm. Um am besten verstehen zu können, in welche Akteure sich die Masken aus dem Design Kapitel aufteilen, werden diese Klassen in den Masken mit blauem Rahmen gekennzeichnet.

In [Abbildung 5.9] sieht man die Aufteilung der `InfoblockForm`, welche die Voransicht des InfoBlocks war. Aus dieser Form werden die anderen Editoren aufgerufen. Die Responce Verwaltung bekommt eine eigene visuelle Klasse - `ResponcesBrowser`. Ebenfalls wird für jede `Element-Preview`, also die Voransicht der Infoblockbestandteile eine generische Komponente erzeugt. Auf der rechten Seite befindet sich das `Details` Pannel, welches ebenfalls eine eigene Klasse bekommt.

In [Abbildung 5.10] sieht man die Maske zum Bearbeiten einfacher Annotations-Ausdrücke. Zu Errinnerung, Fact Annotationen, Effect Annotationen sowie einzelne Bestandteile von Conditions und Deactivate Effect Formlen können mit dieser Maske bearbeitet werden. Eine solche Anweisung an den Reasoner, die durch die Maske generiert wird, nennen wir ab nun Expression. Der `ExpressionBrowser` ist also eine Auflistung von `ExpressionEditoren` mit denen die Reasoner Annotationen wie `ib4 beschreibt Wissenshaft` generiert werden.

Die Maske zu Verknüpfen von Expressions mittels boolischer Operatoren, welche für Deactivate Effect Formeln und Condtion Formeln benötigt wird, kann in nur einer graphischen Klasse geka-

peslet werden - dem `FormulasEditor`.[Abbildung 5.11] Denkbar wäre hier ebenfalls den rechten Bereich, der für die CRUD Operationen, der Conditions oder DeactivateEffects und den linken Bereich, der für die Eingabe der jeweiligen Condition oder DeactivateEffect Formel zuständig ist, ebenfalls in zwei weitere View Klassen zu spalten. Da diese Maske, jedoch immer als eine ganze Einheit verwendet wird und nur aus Standart Flex Widgets besteht, wird für den Prototypen für diese architektonische Verschönerung verzichtet.

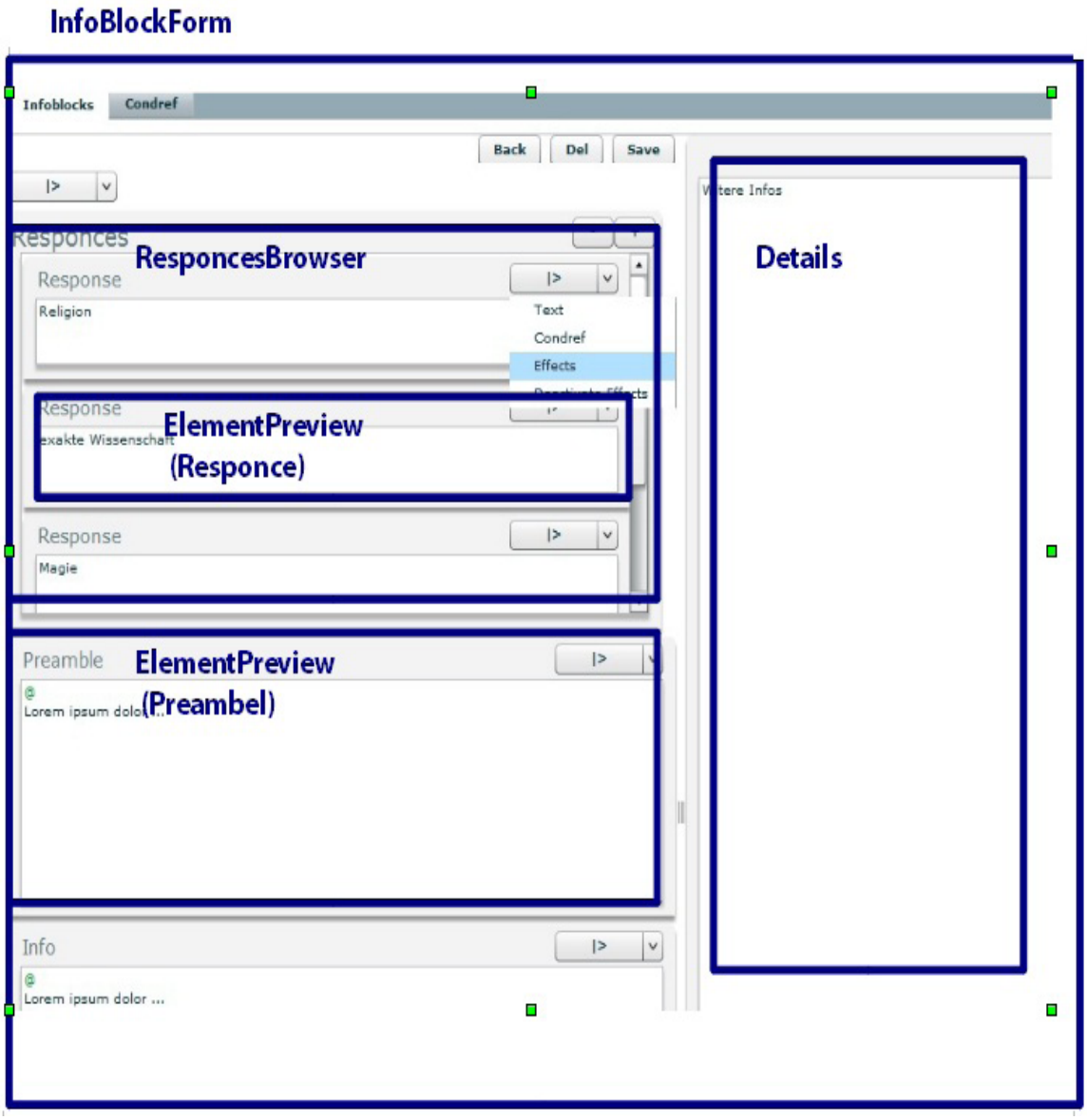


Abbildung 5.9: InfoBlockForm Aufbau

Der `FormulasEditor` hat weiterhin eine Referenz zu dem Expression Editor. Wie man weiß, sind die einzelnen Teilausdrücke, welche durch den Formaulas Editor mittels boolischer Operatoren verknüpft werden semantisch vollkommen identisch mit den Ausdrücken, welche durch den Expression Editor (Facts, UpdateEffects) erzeugt werden (siehe auch [Abbildung 5.11] Tabellenzeilen). Daher wird der `ExpressionEditor` aufgerufen um ein Teilausdruck in der Tabellenzeile des `FormulasEditors` zu erzeugen.

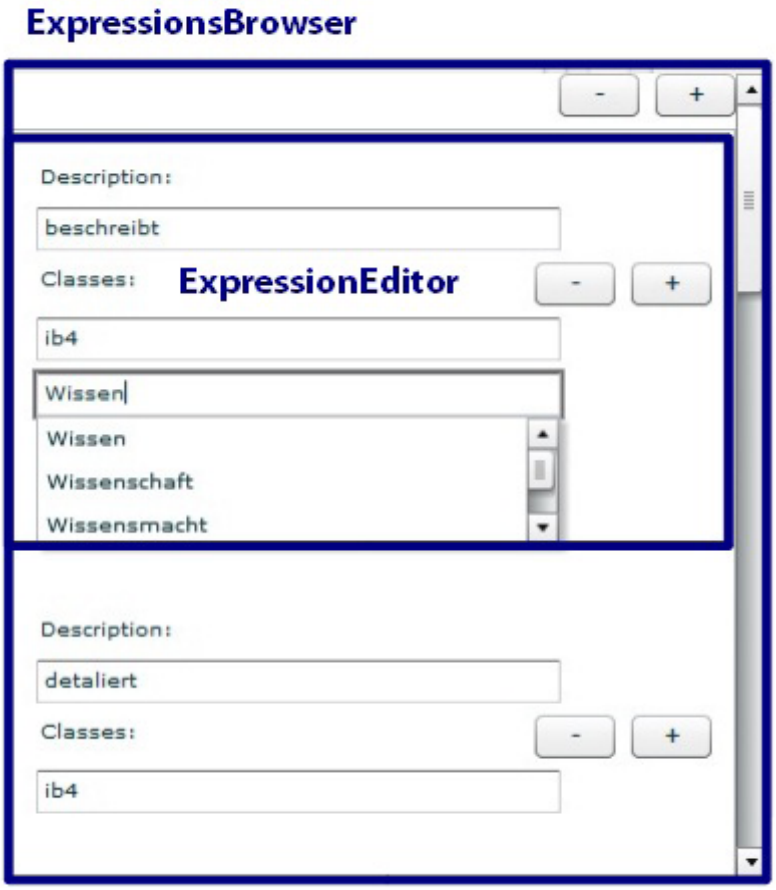


Abbildung 5.10: ExpressionsBrowser Aufbau

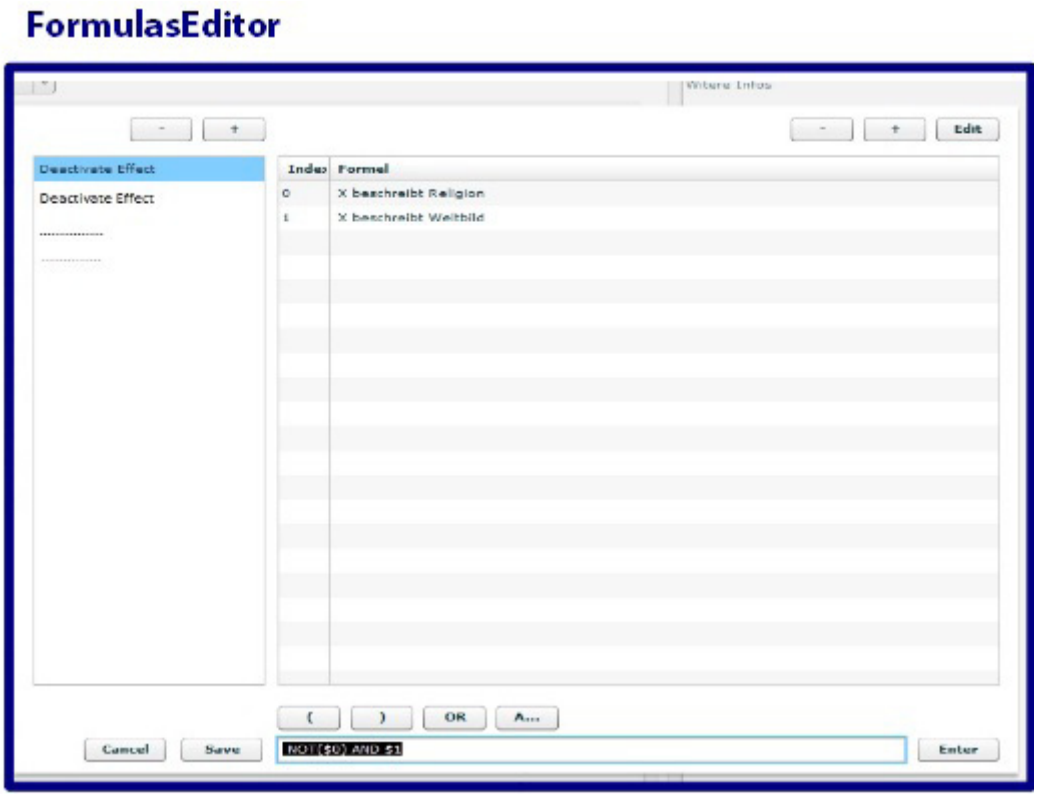


Abbildung 5.11: Formulas EditorAufbau

Möchte man nun die Kopmonenten in der oben dargestellten Weise auf dem Bildschirm Platzieren, so ergibt sich automatisch durch das Flex Framework folgendes Referenzendiagramm:

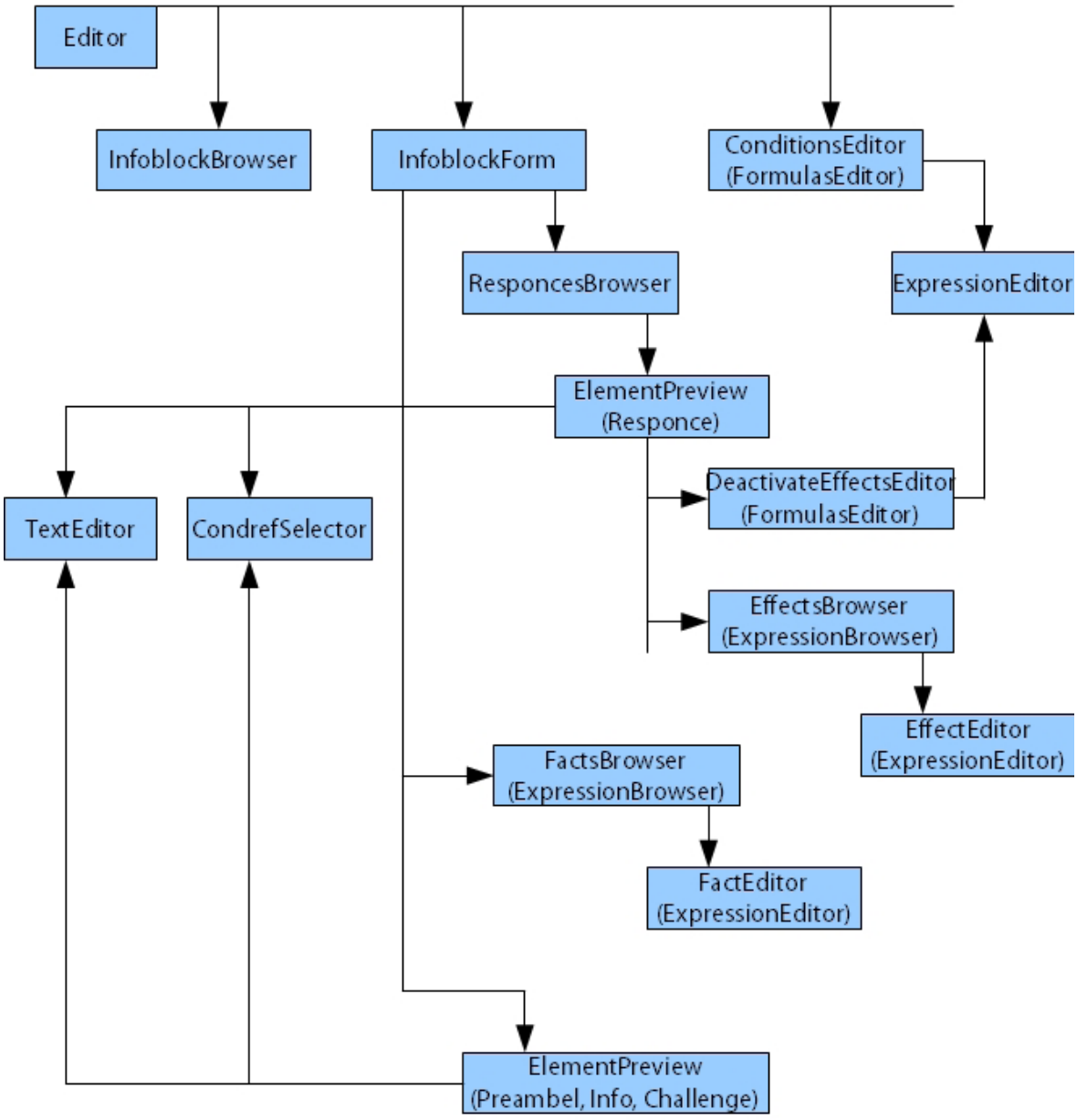


Abbildung 5.12: Editor Views Graph

Diese Klassen aus der Abblidung[5.12] sind also die Views in der Presentation Model Architektur. Sie zeichnen die Masken aus dem Design Kapitel auf dem Bildschirm und befinden sich in der Presentation Schicht. Zusätzlich zu jedes dieser Masken Klassen wird entsprechend eine eigene PM Klasse, welche die Interface Logik der Maske verkapselt, erzeugt. Die PM's selbst kennen sich in dem hierachielosen Ansatz nicht. An dem Beispiel aus dem [5.2.4. Presentation Patterns] erläutert, bedeutet das also, dass das InfoblockBrowserPM das InfoBlockPM nicht kennt. Genaue Implementierung wird in [6.3] erläutert . Weiterhin besitzen einige PM's Referenezen zu der

Applikation Schicht. Sie rufen sog, Delegaten auf um Anfragen an das Backend abzuschicken.

2. Application Schicht
Diese Schicht ist für den Kommunikationsfluss der Anwendung zuständig. In der Diskussion zwischen der hierarhischen und hierachilosen Implementierung des Presentation Model Patterns in [5.2.4] wurde deutlich, dass einige zusätzliche Akteure notwendig sein können um die Koordination zwischen den PM's in der Anwendung zu steuern. Dazu gehören eigene Events, Presenter oder SharedObjects. In dieser Schicht würde man also solche Aktuere platzieren.

In dieser Anwendung gibt es wenige Anwendungsfälle für die Koordination der PM's unter einander, sodass man allein mit dem Messaging Feature mit dem zentralen EvnentBus des Parsley Frameworks auskommt. Dadurch kann die Koordination allein durch Custom Events sichergestellt werden. Beispielhaft sind `ShowInfoBlockEvent` zum Wechseln vom InfoBlockBrowser zu der InfoBlockform und anzeigen des gewünschten InfoBlocks oder `EditElementEvent` zum Einblenden des richtigen Editors für das Annotieren der InfoBlock Bestandteile.

Neben der Kommunikation zwischen den Client Komponenten unter einander wird natürlich ebenfalls eine Kommunikation zum Backend benötigt. Denn die eigentlichen Daten (InfoBlocks, Conditions oder TBox Ontologie) befinden sich auf einer physikalisch anderen Maschine. Akteure welche diese Daten vom Backend abfragen heißen Delegaten und werden ebenfalls in die Applikation Schicht platziert. Die Delegaten wiederum übertragen die Daten Objekte, auch Vertreter Objects (VO) oder Domain Objects genannt. Diese Domain Objects befinden sich wie der Name schon sagt nicht in dieser, sonder in der Domain Schicht (siehe 3.Domain).

Der Prototyp der Anwendung kommt mit drei Delegaten aus:

- InfoBlockDelegate
- ConditionDelegate
- ReasonerDelegate

Die Anzahl der Delegaten hängt stark von den Services im Backend ab, also der Art und Weise welche Datenzugriffe das Backend anbieten kann. Eine genaue Diskussion hierzu befindet sich entsprechend im zweiten Teil des Architektur Kapitels - Backend Architektur. An dieser Stelle sollen kurz die Referenzen zu den Delegaten und der Wokflow der Persistenzschicht betrachtet werden:

Das InfoBlockBrowserPM erhält den Infoblock Nachfolger-Beziehung Graph über den InfoBlock-Delegaten. Ebenfalls vom InfoBlockDelegaten holt das InfoBlockFormPM sich den selektierten

InfoBlock mit allen Unterelementen (Preambel,Info, Challenge, Responces, Facts u.s.w),. Der Zustand des selektiereten InfoBlocks wird von der Anwendung gepflegt, es werden also CRUD Operationen an ihm und seinen Unterelementen ausgeführt. Nach den Änderungen wird schließlich das gesamte InfoBlock von dem `InfoBlockFormPM` wieder über den `InfoBlockDelegaten` an den Server zurückgeschickt.

Der `ConditionEditorPM` bearbeitet die Liste der Conditions über den `ConditionDelegate`.

Der `ReasonerDelegate` wird von dem `ExpreesionEditorPM` genutz um die Vervollständigungen für die korrekten Bezeichner bei der Eingabe der Annotationsausdrücke für Facts, Effects und Conditions zu erhalten.

3. Domain Schicht
In der Domain Schicht befinden sich die Datenkapselungklassen an den die Editierungsarbeit verrichtet wird. Die Dialoganwendung besitz bekanntlich eine Persistenzschicht in der die Daten, hier in Form von XML Files, gespeichert sind. Wenn diese Daten angefragt werden, werden sie im Backend in normale Objekte gekapselt und an das Frontend übertragen. Da das Frontend und das Backend in verschiedenen Programmiertsprachen implementiert sind, ist hier eine Serialisierung notwendig. Dafür sorgt wie im Grundlagen Kapitel beschrieben das BlazeDS Framework.

Es ist aber erforderlich, dass sowohl in ActionSctipt auf der Clientseite, als auch in Java auf der Serverseite identische Datentypen definiert werden. Wenn man von diesen Datentypen auf der Clientseite spricht, so werden diese auch Vertreter Objects(VO) genannt. Man spricht auch von einem Domain Model.

Es sollte bereits ersichtlich sein, welche Domain Objects für diese Anwendung benötigt werden: InfoBlock, Responces, Preambel, Info, Challenge, Effect u.a. Auf diese Aufteilung wird ebenfalls in dem folgenden Abschnitt, Backend Architektur eingegangen.

4. Infrastructure
In dieser Schicht werden die restlichen Utility Klassen der Applikation platziert. Z.B. Die Parser um Condtions und DeactivateEffects Ausdrücke aus der Infix in die Postfix Notation zu überführen und umgekehrt.

5.2.8 Schlusswort

Der Abschnitt Frotnendarchitektur des Architektur Kapitels beschäftigte sich mit der Aufstellung

des Klassengerüstes für den Cleintteil der Anwendung, welches in Flex/ActionScript implementiert wird.

Bevor die tatsächliche Architektur der Anwendung vorgestellt werden konnte, wurden diverse Verfahren aus vergangener Zeit kritisch betrachtet und neue Verfahren vorgestellt. Die Analyse der neuen Verfahren zu Realisierung der clientseitigen Architektur in Flex sprach sich für das Presentation Model aus. Anschließend wurden die Begriffe Inversion of Control/Dependency Injection eingeführt und einer bestimmten Problematik aus der Anwendung gezeigt, dass IoC Frameworks mit wenig Aufwand eine lose gekoppelte Architektur herstellen können. Schließlich wurde die Entscheidung zu dem Presentation Model Pattern in Verbindung mit IoC durch kurze Referenz auf die Adobe Cairngorm 3 Blaupause untermauert und der tatsächliche Klassengerüst der Clientanwendung orientiert an die designten Masken vorgestellt.

5.3.Backend Architektur

5.3.1. Einleiteitung

Der Abschnitt Frontendarchitektur beschäftigte sich mit den Software Engeneering Mustern zu Umsetzung der graphischen Clientanwendung. Es wurde gezeigt welches Klassengerüst benötigt wird, um die Editor Bearbeitungsmasken aus dem Design Kapitel umzusetzen. Die Masken bearbeiten, wie bekannt die Persistenzschicht des Dialogsystems und schaffen Querverbindungen zu der TBox, in Form von Textvervollständigungen bei der Eingabe der Annotations Formeln.

Da die Clientanwendung sich physikalisch auf einer ganz anderen Maschine befindet, als das Dialogsystem, muss das Dialogsystem eine Erweiterungskomponente erhalten, um die Daten aus der Persistenzschicht und der TBox an das Benutzerinterface auf der Maschine des Autors bereitzustellen. Diese Komponente ist das Backend der Editor Anwendung.

In dem Frotendarchitektur Kapitel wurden Delegateklassen vorgestellt. Bis jetzt ist bekannt, dass die Delegaten die notwendigen Daten bereitstellen. Das Backend bietet Services, welche die Delegaten Anfragen von dem Clientrechner empfängt, darauf wenden sich die Services an die Persistenzschicht und den Reasoner mit der TBox und senden schließlich das Resultat an den Clientrechner zurück.

In diesem Abschnitt wollen wir den genauen Aufbau dieser Services in der Backend Komponente betrachten.

5.3.2. Analyse des Datenmodels des Dialogsystems

Man erinnere sich, das Dialogsystem verwaltet die Information in einer Ontologie, welche sich in TBox, für allgemeine Aussagen und ABox, für benutzerspezifische Aussagen unterteilt, sowie in einer Persistenzschicht, hier in Form von XML Files zu Speicherung der Dialoginhalte und Annotationen an die benutzerspezifische ABox.

Das Dialogsystem bietet selbstverständlich Mechanismen zu Operation auf dem Reasoner un der Persistenzschicht an. Es muss ja selbst in der Lage sein mit jeder Benutzerantwort, die InfoBlock XML Files zu lesen, dessen Information in Datenobjekte speichern, um schließlich die Annotationen an den Reasoner senden zu können und die Texinhalte zu HTML Seiten zu verarbeiten. Daher liegt die Schlussfolgerung nahe das gleiche Datentypen-Model, sowohl für das Administrator Interface, also die Editor Applikation, als auch für das Dialogsystem selbst zu verwenden um eine Redundanz zu vermeiden.

Was den Reasoner betrifft, so benötigt der Editor ganz andere Zugriffe als das Dialogsystem selbst. Wie bekannt, operiert das Dialogsystem hauptsächlich auf der ABox des Benutzers. Es sammelt Erfahrung über ihn, in dem es bestimmte ontologische Tatsachendefinitionen in der ABox speichert. Der Editor kann nicht auf der Abox operieren, da diese zum Zeitpunkt der Erstellung des Dialogs nicht existiert und da sie für jeden Benutzer unterschiedlicht ist. Der Editor wird hingegen auf der TBox operieren, welche , bekanntlich, eine Abbildung der realen Welt ist und somit vorschreibt welche Annotationen für die Abox überhaupt erstellt werden können. Somit kann die Reasoner Schnittstelle ganz einfach wiederverwendet werden, indem sie um weitere Abfragen an die TBox erweitert wird.

Im Gegensatz zu der Reasoner Schnittstelle könnte die Editor Applikation theoretisch exakt die gleichen Datentypen für die Zugriffe auf die Persistenzschicht verwenden wie das Dialogsystem selbst. Die Leseoperationen konnten sogar eins zu eins wiederverwendet werden. Denn die Anforderung für den Datenzugriff aus der Persistenzschicht ist sowohl für den Editor, als für das Dialogsystem gleich. Die InfoblockForm aus dem Frontendarchitektur Abschnitt und eine Dialogseite sind im Grunde zwei Unerschiedliche Ansichten auf die gleichen Datentypen. Zusätzlich würde man dann diese Schnittstelle um die Schreib und Löschoperationen erweitern.

Bei genauer Betrachtung stellt man jedoch fest, dass das von Alexander Bokov konstruierte Datenmodel leider nicht exakt so für die Editor Erweiterung verwendet werden kann. Diese Problematik ergibt sich größtenteils aus der Tatsache, dass die Datentypen, welche die Persistenzscshicht abbilden für den Editor vom Backend zum Frontend übertragen werden müssen, wobei sie zu einem Übertragunsformat serialisiert werden. Folglich soll dieses Problem genauer

betrachtet werden.

Hindernd für die Übertragung der Datentypen erweist sich die Tatsache, dass das Datenmodel des Dialogsystems nicht genügend von dem Persistenzmechanismus - XML - abstrahiert. Alexander Bokov verwendet eine sog. Document Object Model (DOM) Bibliothek zum abbilden von XML Files auf Objekte. Dabei wird das XML Dokument zunächst als eine abstrakte Datenabbildung in den Speicher geladen. Diese Datenabbildung orientiert sich an der XML Sprache und ist also für alle Typen von Dokumenten gleich. Die Datentypen des Dialogsystems speichern immer eine Document Object Model Abbildung des entsprechenden XML Files in einer Instanzvariable und bieten entsprechende Methodenzugriffe auf diese XML File Abbildung an.

Dieses Verhalten kann man folglich an dem InfoBlock Beispiel erläutern:

```
<InfoBlock name=1>
       <Responce> Antwort A </Responce>
       <Responce> Antwort B </Responce>
       <Preambel> Hier beginnt ein neues Thema </Preambel>
       <Info> Diese Information ist sehr komplex</Info>
       <Challenge> Was möchten Sie als nächstes erfahren ?</Challenge>
       <successor-list>
              <succ> InfoBlock 2</succ>
                           .
                           .
                           .
       </successor-list>
</InfoBlock>
```

Oben sieht man eine etwas vereinfachte InfoBlock XML Implementierung. Das Dialogsystem definiert dazu eine InfoBlock Klasse. Diese speichert in der Instanzvariable `data` ein DOM Model des InfoBlocks. Das DOM Model ist abstakt. Es kommt selbst mit einfachen Datentypen aus, die auf alle XML Dokumente angewendet werden können. Es gibt also nicht etwa ein Datentyp Responce, sondern ein Datentyp Element, welches als Namen „Responce“ trägt. Ferner sagt das DOM Model über den InfoBlock nicht aus, das er Responces besitzt, sonder es sagt allgemein - Ein InfoBlock besitzt Kinerelemente. Diese Kinderelemente sind alle Tags unter dem `<InfoBlock>` Tag. Der Datentyp InfoBlock aus dem Dialogsystem bietet nun für einen leichteren Zugriff, die Methode `getResponces` an, welche auf das DOM Objekt zugreift und mittels eine XML Abfrage Sprache `XPath` das DOM durchsucht und die Responces des InfoBlocks zurückliefert. Der folgende Java Pseudocode veranschaulicht diese Methode.

```
Class InofoBlock{
   data:Document; //DOM des InfoBlock XMl Files

   public List<Responce> getResponces(){

      List<Responce> ret = new ArrayList<Responce>();

      //Xpath query
      Nodes nodes = data.query(„//Responce“);
      for (int i = 0; i < nodes.length; i++) {
           ret.add(Reponce.createFromNode(nodes[i]));
      }
      return ret;
   }
}
```

Diesem Verhalten ist zunächst wenig entgegenzusetzten wenn man auf einer Maschine arbeitet, jedoch ergeben sich Nachteile, wenn man solche Datentypen von einem Rechner zum anderen übertragen möchte, wie es die Editorapplikation erfordert. Das von Adobe mitgelieferte Framework BlazeDS sorgt für eine automatische Übertragung der Datentypen von Java zu ActionScript und umgekehrt, in dem es diese in das binäre Format AMF serialisiert, überträgt und auf der anderen Seite wieder deserialisiert. Dabei kann nur der Zustand des Objekts, also der Inhalt seiner Instanzvariablen serialisiert und übertragen werden, jedoch nicht das Verhalten, also seine Methoden. D.h. natürlich nicht, dass diese Objekte gar keine Methoden mitbringen können, man muss Sie nur auf beiden Seiten der Anwendung -Frontendclient(ActionScript) und Backendserver(Java) - implementieren.

Würde man also so ein Datentyp, welcher intern auf einem DOM des XML Files basiert vom Backend zum Frontend übertragen, so würde man alle Funktionen der Java XML DOM API, sowie die von Alexander Bokov implementierten Zugriffsmethoden in den Datentypen nach der Übertragung verlieren. Man müsste also die Zugriffsmethoden auf der Clientseite nachimplementieren ohne dabei die Hilfsfunktionen der DOM Bibliothek verwenden zu können, da ja diese ebenfalls nicht serialisiert und übertragen werden können.

Dieses ist aber nicht der einzige Nachteil. Ein weiterer Nachteil ist, dass bei dieser Variante zu viele unnötige Objekte und deren Zustände serialisiert und übertragen werden, nämlich die ganzen abstrakten Datentypen aus der DOM API. Dieses kann sich durchaus auf die Reakti-

onsgeschwindigkeit der Anwendung auswirken. Und schließlich ist eine Bindung an das XML Format in der ganzen Applikation nachteilig, da der Persistenzmechanismus schwieriger ausgetauscht werden kann. Dass ein Austausch durchaus Sinn macht, wird sich an einer späteren Stelle zeigen.

Benötigt wird also ein Datenzugriffsmechanismus, welcher von der Persistierungsmethoden abstrahiert und nur die notwendigen Daten vom Server zum Client überträgt. Eine einfache Methoden für so einen Mechanismus wird im nachfolgenden Abschnitt betrachtet.

5.3.3. Data Access Object Pattern

Im vorherigen Abschnitt wurde gezeigt, dass eine Abstaktion von dem Persistenzmechanismus und leichtgewichtige Datentypen für die Übertragung vom Backend zum Frontend notwendig sind.

Eins der Standartrezepte für diese Aufgabe ist das Data Object Access Pattern (DAO Pattern). Das DAO Pattern erlaubt eine Abstaktion von dem tatsächlichen Persistenz Mechanismus. [SDN10] erläutert das DAO Pattern. Dieses soll nun auf die Anwendung übertragen werden. Im folgenden wird das DAO Pattern an dem `InfoBlockService` und zwe Methoden `getInfoBlock(String id)` zum Finden eines bestimmten InfoBlocks und `saveInfoBlock(Infoblock ib)` zum Speichern eines bestimmten InfoBlocks erläutert. [Abbildung 5.13]

Das Prinzip ist im Grunde Simpel. Die Logik für den konkreten Zugriff auf die Persistenzschicht (hier also die Arbeit mit den XML Files) wird in das DAO Objekt ausgelagert, welches ein definiertes Aufruf Interface nach Außen implementiert. Das DAO übersetzt die Daten aus der Persistenzschicht in normale Objekte.

In diesem Beispiel übernimmt also das `InfoBlockXMLDAO` den Zugriff auf das InfoBlock XML File. Wenn seine Methode `saveInfoBlock(InfoBlock ib)` aufgerufen wird, so sucht das DAO ein File mit dem gleichen Namen, wie die id des InfoBlock's und überschreibt es mit den Daten aus dem übergebenen InfoBlock Objekt bzw. legt ein neues an, falls es nicht existiert. Der Service selbst muss sich nicht darum kümmern, wie das InfoBlock gespeichert werden. Er erledigt diese Arbeit über das DAO. Somit kann das DAO, da es ein fest definiertes Interface implementiert leicht ausgetuscht werden.

Eine weitere Eigenschaft dieses Patterns ist das Stützen auf Domain Objekte. Wie man deutlich sieht wird die Information aus dem InfoBlock XML File in der InfoBlock Klasse verkapselt. Die InfoBlock Klasse selbst hat nichts mit der XML Datenstruktur zu tun. Das DAO speichert in dem

Domain Objekt nur die notwendigen Informationen und gibt dieses an den Aufrufer zurück. Dies kommt der Editor Appliaktion entgegen, da man genau dieses Objekt an den Client übertragen kann.

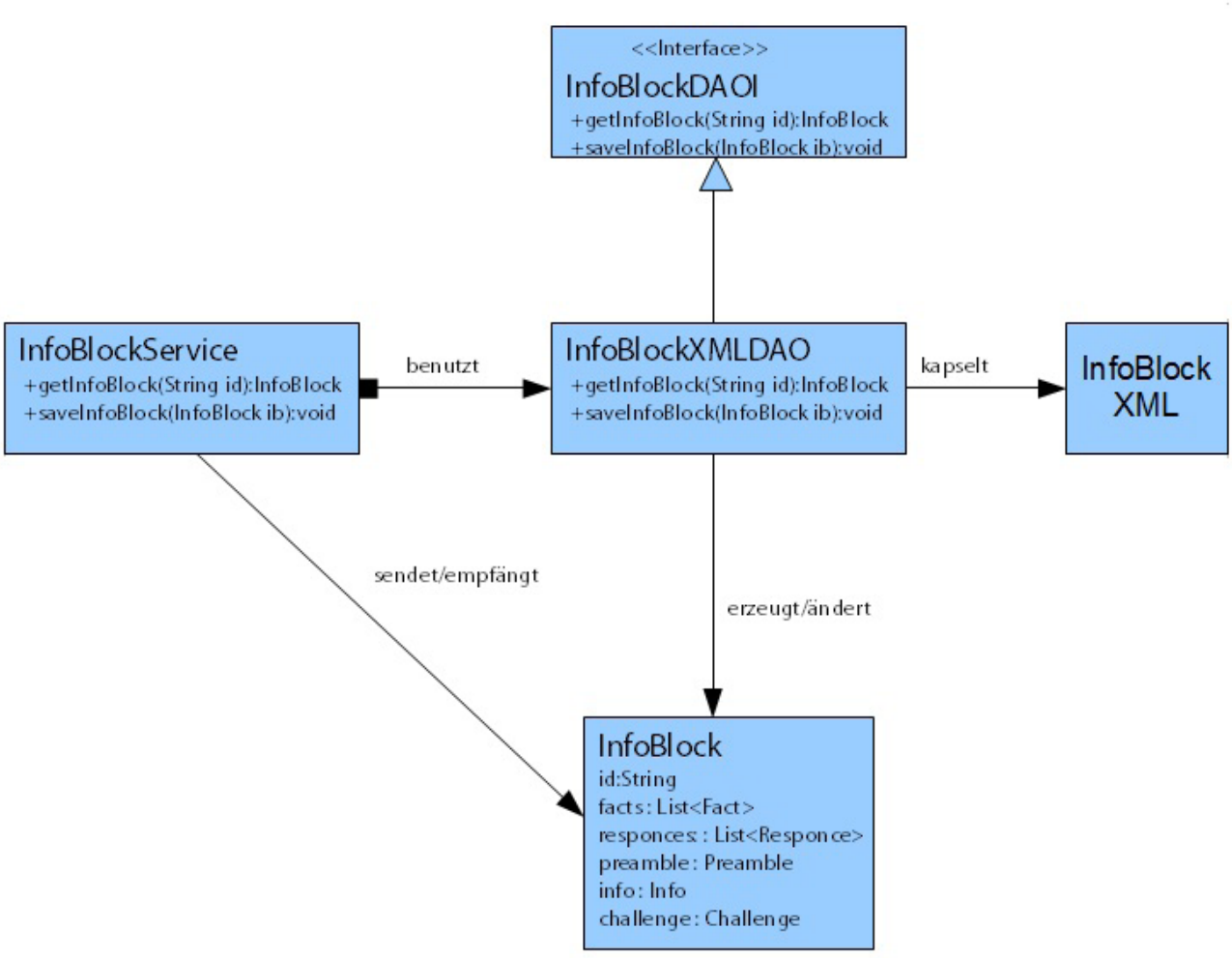


Abbildung 5.13: DAO Pattern

Das DAO Pattern ist also eine simple Vorgehensweise um den Zugriff auf die tatsächliche Persistenzschicht zu kapseln. Es wird jedoch eine händische Implementierung der Zugriffe auf das XML File erfordert. Dafür kann natürlich wieder die DOM Bibliothek verwendet werden, jedoch wird sich die Kenntniss über das DOM nicht mehr in der InfoBlock Klasse sondern in der InfoBlockXMlDAO Klasse befinden.

5.3.4. Architektur Umsetzung

In diesem Abschnitt werden die wichtigsten Akteure im Backend vorgestellt und der Kommunikationsprozess zu dem Frontend erläutert.

Aus dem Frontendarchitektur Abschnitt weiß man, dass im Forntend bestimmte Kommunika-

tionsschnittstellen zum Server existieren - die Delegaten. Die Gegenspieler zu den Delegaten sind Services im Backend. Das BlazeDS Framework erlaubt, wie im Grundlagenteil dargestellt, eine beinahe vollkommene Transparenz, bei der Implementierung dieser Delegaten und Services. Die Bezeichnug, Remote Procedure Call ist eine viel treffenderer Begriff für diese Technik.

Dabei wird ein Service im Backend wie ein normales Objekt implementiert. Service Objekte besitzen ganz normale Funktionen mit Datenparametern. Auf der Clientseite im Frontend, kann dieser Service durch den Delegaten wie eine lokale Funktion aufgerufen werden. In Wirklichkeit wird aber kein lokaler Aufruf gemacht sondern ein Remote Procedure Call, wodurch eine identischer Funktion im Service Objekt im Backend ausgeführt wird. Der Return Wert dieser Funktion wird automatisch an den aufrufenden Rechner serialisiert übertragen.

Die Aufgabe für das Backend besteht also darin, Services bereitzustellen, welche auf die Persistenzschicht und TBox zugreifen, die dort befindenden Daten in Domain Model Objekte kapseln und diese an den Delegaten zurückgeben, oder umgekehrt neue oder geänderte Objekte in der Persistenzschicht ablegen. Die Arbeit auf der Persistenzschicht selbst übernehmen die DAOs. Somit ist die Architekturumsetzung auf der Serverseite trivial. Analog zu den vorgestellten Delegaten im Frontend, ReasonerDelegate, InfoBlocksDelegate und ConditionsDelegate werden drei Services im Backend defeniert - ReasonerService, InfoBlockService und ConditionsService.

Die Services erlauben es die Daten aus der Persistenzschicht und der TBox abzufragen, indem sie die Anfrage sofort an die Data Access Objects weiterleiten. Zu jedem Service gibt es also wiederum ein DAO, der den Zugriff auf die Daten kapselt- `InfoBlocksXMlDAO`, `ConditionsXMLDAO` und `ReasonerRacerDAO`. Diese Data Access Objects verwenden keine anderen Objekte aus der Dialogsystemappliaktion, da es aus den zuvor angeführten Gründen nicht sinnvoll ist. Um eine Redundanz zu vermeiden sollte später in Betracht gezogen das Dialogsystem so zu refakotorisieren, dass es ebenfalls das DAO Pattern und somit, dass gleiche Datenmodel wie der Editor verwendet. Schließlich gibt es zu jedem Objekt aus der Frontend Domain Schicht, ebenfalls ein Domain Datentyp auf der Backend Seite. Wie bereits zuvor angeführt, wird dabei versucht, jeder wichtigen Annotation einen eigenen Datentyp zu geben. Mann erzeugt z.B. ein eigenes Preambel Objekt oder Challenge Objekt und nicht etwa Element Objekte mit dem Namen „Preambel" oder „Challenge". Dieses ist mit einem durchaus übershaubaren Mehrraufwand verbunden, erlaubt aber eine zusätzliche Fehlersicherheit.

Im Frontendarchitektur Abschnitt wurden bereits einige dieser Datentypen aufgeführt, an dieser Stelle wird das Datenmodel vollständig erläutert. Zunächst gibt es eine InfoBlock Datentyp, welcher wiederum Responces, Preamble, Info, Challenge und Facts besitz. Die Responces besitzen, UpdateEffects und Deactivate Effects. Die Condition Abfragen erhalten ebenfalls ein

eigenen Datentyp - Condition.

In Analyse Kapitel haben wir gesehen, dass die Annotation Ausdrücke in den Effects, Facts und Conditions eine semantische Ähnlichkeit haben. So sind die Ausdrücke in den Facts und UpdateEffects, sowie in den Conditions und DeactivateEffects semantisch gleich. Ferner setzen sich die Condition und DeactivateEffects Ausdrücke sich aus den Ausdrücken in den Facts und UpdateEffects zusammen. Diese Beziehung kann objektorientiert durch Vererbung und Aggregation modelliert werden. Facts und UpdateEffects werden somit vom Datentyp Expression abgeleitet und UpdateEffects und Facts vom Datentyp Query. Eine Query setzt sich aus mehreren Expressions zusammen.

Expressions können bekanntlich wie folgt ausssehen:

```
Benutzer Wissenschaft interessiert
```

oder

```
Interesse groß
```

Somit bekommt der Expression Datentyp ein Description Property, in das Assoziationsbezeichner wie groß oder interessiert gespeichert werden und ein Array für Objektbezeichner wie `Benutzer`, `Wissenschaft` oder `Interesse`. In dem Array befinden sich je nach Situation zwei oder ein Objektbezeichner.

Eine Query Kombiniet die Expression Ausdrücke, fragt also ob diese gelten oder nicht.

```
Benutzer Wissenschaft interessiert AND Interesse groß
```

Hierfür führen wir ein Array von Expression Objekten und ein String Property `formula` für den boolischen Ausdruck ein, welcher die Expressions anhand der Indizes im Array verknüpft.

```
Query => {
       expressions:Array => {
                            1 => Expression { Benutzer Wissenschaft interessiert }
                            2 => Expression { Interesse groß }
       }
                 formula:String => „$1 AND $2“
}
```

Schließlich bleibt noch einen Format für die Daten zu bestimmen , welche der ReasonerService

liefert. Da die Anfragen für den Reasonerservice in den Prototypen einfache Textvervollständigungen sind, kommt die Anwendung hier erst einmal mit simplen Array Strings aus. Angenommen der Benutzer möchte die Expression `Interesse groß` eingeben. Fokussiert er nun das Feld für die `description` Property der Expression, also das Feld wo später `groß` stehen wird, so wird über den `ReasonerService` eine Liste der möglichen Bezeichner für diesen Typ der Expression angefordert. Man sieht also, dass simple String Listen zunächst vollkommen ausreichend sind.

In diesem Abschnitt wurden die Services und die Datentypen im Backend betrachtet. Die Backend Seite der Anwendung enthält deutlich weniger Logik, da die gesamte Zustandverwaltung vom Frontend übernommen wird. Es wäre aber durchaus ein Szenario denkbar, bei dem dem Backend mehr Bedeutung zukommt. Vor allem die Arbeit auf der Persistenzschicht könnte in feiner granulare Services gegliedert werden, wodurch die Zustandsverwaltung im Frontend erleichtert würde. Warum dieses sich bei dem gewählte Persistenzformat schwierig erweist, erläutert das anschließende Kapitel.

5.3.5. Nachteile Des Persistenzformats XML für die Editotapplikation

Das Dialogsystem speichert die Dialoginhalte im XMl Format. Für das Dialgosystem selbst ist das vollkommen ausreichend, da diese Inhalte nur gelesen werden. Es wird immer mit einer Benutzerantwort ein einzelnes InfoBlock XMl File geladen. Die Anwendung durchsucht weder mehrere Files auf ein Mal, weder verändert sie diese. Das ganze sieht aber für die Editor Appliaktion anders aus. Dieser Abschnitt wird zeigen, wieso einige durchaus denkbare Features für die Editorapplikation mit dem XMl Persistenz Format nur schwer umgesetzt werden können und eine Anregung liefern entweder eine relationale oder dokumentorientierte Datenbank zu verwenden.

Größtenteils betrifft die Diskussion die Editierung des InfoBlocks, also der Dialogseiteninhalte. Es ist bekannt, dass ein InfoBlock aus mehreren Bausteinen besteht - Preambel, Info, Challenge und Responces. Den Responces werden weitere Annotationen zugeordnet - Update und Deactivate Effects. Ebenfalls kann der InfoBlock mehrere Facts zugewisen bekommen. Es sind also CRUD Operatioenen an allen diesen Unterelementen des InfoBlocks erforderlich.

Momentan wird der InfoBlock folgender Maßen editiert:
Der komplette InfoBlock wird vom Server Backend in die Clientanwendung geladen und sein Zustand wird während der Sitzung gepflegt. Der Autor erledigt also die CRUD Operationen an dem InfoBlock und seinen Unterlementen. Nach Fertigstellung der Arbeit an einem InfoBlock, speichert er den gesamten InofBlock, wodurch der geänderte InfoBlock wieder zurück zum

Server übertragen wird und dort seine vorherige Version überschreibt.

Die Frage lautet: Wieso gibt es nicht einen eigenen Delegaten und Service für jedes Unterelement (oder zumindest jedes wichtige) Unterelement des InfoBlocks? Der Workflow würde wie folgt aussehen: Bei jeder Änderung, Erzeugen, Löschen, Bearbeiten von InfoBlock Elementen wird nicht mehr lokal der Zustand des InfoBlock Domain Objekts verändert, sondern immer ein Befehl an ein entsprechendes Backend Service durch einen Delegaten gesendet. Es gibt für jede Änderung eines Element im InfoBlock eine eigene Service Methorde. Somit wird nicht mehr nach Fertigstellung der Arbeit am InfoBlock das gesamte Objekt übertragen, sondern die einzelnen Unterobjektänderungen werden sofort an das Backend mitgeteilt. Möchte man also eine neue Repsose zum InfoBlock `ib1` hinzufügen, so sendet man sofort an den Server die Nachricht *„füge die neue Responce zu ib1 hinzu"* oder im Fall einer Änderung *„aktualisiere die Responce x von ib1„*. Das Backend findet den gewünschten InfoBlock in der Persistenzschicht und aktualisiert dann das entrpechende Unterelement des InfoBlocks, hier also die einzelne Responce. Nach son einem Befehl synchronisiert man einfach sofort das Interface mit dem Serverzustand. Also bei einem Hinzufügen einer Repsonce - neue Responce in der Persistenzschicht zum InfoBlock hinzufügen und sofort alle Repsonces des InfoBlocks aktualisieren.

Somit kann auf die Zustandsverwaltung des InfoBlocks in der Clientappliaktion über mehrere Masken verzichtet werden, wodurch der Client mit weniger Abhängigkeiten zwischen den Masken selbst auskommt. Außerdem würde es einer zukünftig denkbaren Anforderung nach parallel arbeitenden Autoren entgegen kommen. Wenn mehrere Benutzer Responces hinzufügen, würden sie so sofort die Änderungen von einander mitbekommen. Außerdem ist die Konfliktgefahr zwischen zwei Benutzern geringer. Zwei Benutzer können problemlos an einem InfoBlock arbeiten, so lange sie verschiedene Teilbereiche des InfoBlocks editieren. Bei der ersten Variante ist dies nicht möglich, da derjenige Autor, der zuletzt die Änderungen speichert, somit alle Änderungen des Anderen überschreibt.

Um so eine feinere Granularität für die CRUD Operationen zu ermöglichen erfordert es aber, dass jeder Datensatz eine eindeutige Kennzeichnung, also eine ID bekommt. Denn ansonsten kann man nur mit Array Positionen arbeiten. Ein zweiter Benutzer könnte aber die Reihenfolge der Elemente verändern, solange der erste noch ein Element editiert. Ein Beispiel wäre z.B. Der Wunsch nach einer anderen Reihenfolge der Responces auf einer Dialogseite. Speichert der erste Benutzer nun die Repsponce danach ab, so ersetzt er eine völlig Andere als er zuvor ausgewählt hat. Ebenfalls muss man sich selbst um eine Konfliktauflösung der parallelen CRUD Zugriffe auf das gleiche Element kümmern, wenn man XML als Persistenzmechanismus verwendet. Wenn also ein Benutzer die Preambel und ein Weiterer die Challenge des InfoBlocks ändert, so muss man auf der Backend Seite ein Mechanismus implementieren, der dafür sorgt dass

diese Operationen auf dem XML File nach einander ausgeführt werden, auch wenn sie gleichzeitig eintreffen, damit die Änderungen beider Benutzer übernommen werden. Eine Datenbank nimmt diesen Aufwand ab. Tatsächlich wird eine so feine Granularität für mehrere Autoren momentan nicht gefordert, so dass das Concurrency Szenario nicht weiter in Detail betrachtet wird. Dennoch würde eine Datanbank eine Erleichterung für die Editorrealisierung bringen, da dadurch ohne großen Mehraufwand auf der Backendseite, wie bereits erwähnt zumindest die Pflege des Zustandes des InfoBlocks über mehrere Masken vermieden werden könnte. Außerdem wären die InfoBlocks Annotations durch Indexierung und Auslagerung in separate Tabellen oder Dokumente, besser durchsuchbar und bei der Unterstützung der Formeleingabe für die Annotation konnten so alle

5.4. Test

Dieser Abschnitt beschäftigt sich mit dem Test der Anwendung.

Der Test der Anwendung sollte im Clientbereich angesetzt werden. Da der Client das Backend aufruft, wird dessen Funtktion dadurch mitgetestet. Zu besseren Fehlerfindung, können natürlich die Services im Backen eine zusätzlichen UnitTest erhalten.

Das Frontend ist eine graphische Anwendung, dessen Test immer mit Schwierigkeiten verbunden ist. [Rein08] zeigt wie mit FlexUnit und der Flash Automation API, graphische Komponenten dennoch getestet werden können. Es werden jedoch enorme Schwierigkeiten deutlich. Graphische Komponenten automatisiert zu bedienen erfordert nicht nur hohen Aufwand, sonder ebenfalls eine genaue Kenntniss der Lifecycles dieser in der Applikation. Oft sind Reaktion nach Clicks asynchron und eventbasiert, so dass unklar ist, wann Daten in die Grafiken eintreten.
„... such designing the test to focus just on the component's logic becomes something of an art."
*[*Rein08, Post Creation Testing*]*

Gerade aus diesem Grund wurden die Presentation Pattern untersucht. Das gewähle Presentation Model Pattern erlaubt die wichtigste Logik der graphischen Komponenten zu testen, da diese in die PM Klassen ausgelagert war. Außerdem wurde ein Dependcy Injection Framework verwendet um die Komponenten zu verbinden. Der Test ganz einfach stattfinden, indem die PM Klassen Unit getestet werden. Zusätzlich könnet man im Test das Parsley Framework ebnfalls wie in der Anwendung initialisieren und das Zusammenspiel der PM's testen.

5.5. Zusammenfassung

Im Architektur Kapitel wurde das Konzept für die Anwendung aufgestellt. Dabei gliederte sich

die Analyse analog zu der Anwendung in Frontend, für die clientseitige Applikation und Backend, für die serverseitige Applikation. Im Backend wurde entschieden das DAO Pattern zu verwenden und die Granularität der Service gemäß den Hauptdatenformaten zu halten. Dadurch wurde beispielweise der gesamte InfoBlock vom Backend an das Frontend übertragen, wodurch die Pflege seines Zustands über mehrere Dialogmasken notwendig wurde. Um eine lose gekoppelte Kommunikation zwischen den einzelnen Masken im Frontend sicherzustellen wurde ein Dependency Injection Framework namens Parsley verwendet.

6. Realisierung

Nach dem das Interfacedesign und die Architekturrichtlinie für die Editor Anwendung definiert wurden, werden hier ausgewählte Aspekte der Realisierung betrachtet.

6.1. Remote Procedure Call

Im Architektur Kapitel wurde mehrmals darauf hingewesen, dass eine Kommunikation zwischen dem Frontend und Backend benötigt wird. Das Frontend stellt hierfür Delegate Objekte bereit, welche mit entsprechenden Service Objekten im Backend kommunizieren. Diese Kommunikation wird nun am Beispiel des InfoBlocksDelegate und des InfoBlocksService erläutert und der Webservice-Funktion `getInfoBlock(String id)` erklärt.

Zunächst soll der Backend Service in Java betrachtet werden. Der Service ist ein gewöhnliches Java Objekt.

```
package de.tk.editor.service;
class InfoBlocksService{
   dao:InfoBlockDAOI = new InfoBlockXMLDAO();
   public InfoBlock getInfoBlock(String id) {
       return dao.getInfoBlock(id);
   }
}
```

Die Methode `getInfoBlock()` erwartet einen id Parameter, übergibt den Aufruf sofort an das DAO und gibt das gefundene InfoBlock über den Return-Wert zurück.

Nun muss dieses gewöhnliche Java Objekt als ein BlazeDS Webservice konfiguriert werden. Wenn das BlazeDS Framework richtig in die Anwendung integriert ist, findet sich in der Applikation ein Verzeichnis, namens „flex“, in dem sich mehrere Konfigurationsdateien befinden. Der Webservice wird in der Datei `remoting.xml` konfiguriert. Hier wird eine eindeutige ID defeniert, unter der der Webservice erreichbar wird und der Service Klassenname zu dieser ID eingetragen.

```
<destination id="InfoBlocksService">
       <properties>
              <source>de.tk.editor.service.InfoBlocksService</source>
              <scope>application</scope>
       </properties>
```

```
</destination>
```

Zusätzlich wird dem Service einen sog. Scope gemäß der Java Servlet API für Webanwendungen vergeben. `Application` bedeutet, dass der `InfoBlockService` in der Applikation nur ein einziges Mal instanzeiert wird. Auch wenn es mehrere Autoren an unterschiedlichen Clientrechnern gibt, werden ihre Anfragen immer von einer und der selben Objektinstanz im Backend verarbeitet. Dieses wäre z.B. wichtig, falls man später Backendressourcen, wie InfoBlocks vor parallelem Zugriff mittels Semaphoren schützen möchte. Vergibt man dieses Attribut nicht, so würde mit jeder Anfrage eine neue Instanz des Services erzeugt (Request Scope).

Somit ist der Service unter dem Namen `InfoBlocksService` erreichbar. Im Flex Frontend kann dieser Service nun aufgerufen werden. Hierfür bietet das BlazeDS Framework eine RemoteObjekt Klasse. Die Remote Aufrufe sind immer asynchron. Man muss zu jeder Remote-Funktion eine Callback Funktion, auch Clousure genannt, angeben. Somit wird der Programmfluss nach dem Aufruf fortgesetzt. Das Interface erstarrt also nicht solange es auf die Daten wartet. Wenn die Daten empfangen worden sind, wird die Callback Funktion aufgerufen.

Damit der Aufruf richtig funktioniert, muss also das RemoteObjek richtig initialisiert werden. Diese Initialisierung und die Arbeit auf dem RemoteObject wird in den Delegaten Klassen verkapselt. Damit die Delegaten von mehreren Presentation Models aufgerufen werden können, soll die Callback Funktion dynamisch an den Aufruf übergeben werden. Dieses kann mit einem sog. AsyncToken erreichen.

InfoBlocksDelegate.as :

```
class InfoBlockDelegate {
     //Konstruktor
     perivate var service:ReoteObject;
     public function InfoBlockDelegate(){
          this.service= new RemoteObject();
          this.service.destination = „InfoBlocksService“;
     }
     public function getInfoBlock( id:String, resultHandler:Function,
         faulHandler:Function ):void
     {
         //Führe Remote Aufruf aus
         token:AsyncToken = service.getInfoBlock.send(id);

         //Füge Callbacks für das Aufrufresultat hinzu
```

```
            token.addResponder (new AsyncResponder(resultHandler, FaultHandler));
        }
}
```

Der InfoBlocksDelegate erzeugt ein RemoteObjekt im Konstruktor und speichert es in der Instanzvariable `service`. Dabei wird das Property `destination` des RemoteObjects auf den, im Backend definierten Namen, `InfoBlocksService` gesetzt. Durch `service.getInfoBlock.send(id)` wird der Remote-Aufruf zu der Java InfoBlockService Methode `getInfoBlock(String id)` getätigt. Das Resultat ist erstmal ein AsyncToken. Zu diesem Token werden die Callback-Methoden hinzugefügt, damit diese nach dem Beenden der Übertragung ausgeführt werden.

Nun kann der InfoBlockDelegate vom dem `InfoBlockFormPM` verwerndet werden.

```
Class InfoBlockFormPM {

[Inject] //Dependecy Injection durch Parsley
public var infoBlocksDelegate: InfoBlocksDelegate;
[Bindable]
public var infoBlock:InfoBlock;

    public function displayInfoBlock(id:String){

        //Erzeuge Result Clousure
       resultHandeler = function (event:ResultEvent){
          this.infoBlock = event.result;
       }
      faultHandeler = ...
      this.infoBlocksDelegate.getInfoBlock(id, resultHandler, faultHandler);

  }

}
```

6.2. Lose Kopplung durch Messaging

Im Frontendarchitektur Abschnitt wurde das Presentation Model Pattern anhand von einem vereinfachten Szenario aus der Applikation veranschaulicht [5.2..4]. Dabei gibt es zwei View Komponenten . `InfoBlockBrowser` - eine Auswahl der InfoBlock ID's und `InfoBlockForm` - die Dar-

stellung des ausgewählten `InfoBlocks`. Um das Presentation Model Pattern leichter zu erklären, wurde angenommen, dass beide Views gleichzeitig auf dem Bildschirm dargestellt werden und dass alle InfoBlocks mit den kompletten Daten sofort geladen werden, was in der tatsächlichen Implementierung nicht der Fall ist. Außerdem wurde das Szenario an der hierachischen Variante des Presenation Model Patterns dargestellt, bei der sich die PM gegenseitig kannten.

Tatsächlich ist das Szenario etwas komplexer. Da es nicht genügend Platz auf dem Bildschirm gibt, wird der `InfoBlockBrowser` und die `InfoBlockForm` auf zwei Bildschirmabfolgen verteilt. Der Benutzer sieht also zunächst nur den `InfoBlockBrowser`. Wählt er darin eine ID aus, so wird der `InfoBlockBrowser` ausgeblendet und die `InfoBlockForm` wird gezeigt. Außerdem wird das komplette InfoBlock erst dann geladen, wenn es gezeigt werden soll. Der `InfoBlockBrowser` bekommt also nicht die kompletten `InfoBlocks`, sondern nur eine Liste mit ID's. Genauer gesagt handelt es nicht um eine Liste, sondern um einen Graph, welcher wie bekannt, bis auf ein Paar Referenzen-Zyklen einen baumartigen Charakter hat und somit durch eine Flex Tree Komponente dargestellt wird.

Nun soll dieser Anwendungsfall in der tatsächlichen Implementierung betrachtet werden. Dabei wird wie zum Schluss von [5.6.2.] erläutert eine hierachilose Presentation Model Pattern Variante gewählt. Die einzelnen PM's kennen sich also nicht und kommunizieren durch das Messaging Feature von Parsley.

Demnach gibt es 3 View Komponenten - `Editor`, der Hauptcontainer die Appliaktion, `InfoBlockBrowser` und `InfoBlockForm`. Zur einem Zeitpunkt ist entweder der `InfoBlockBrowser` sichtbar oder die `InfoBlockForm`. Dieses kann mit dem Flex `ViewStack` Widget erreicht werden.

Editor.mxml:

```
<mx:Canvas>
   <mx:Script><![CDATA[

       [Bindable]
        public var model: EditorPM;

    ]]></mx:Script>

   <parsley:FastInject property="model" type="EditorPM">
```

```
    <mx:ViewStack selectedChild = {model.selectedChild} >
        <presentation: InfoBlockBrowser id="infoBlockBrowser"/>
        <presentation: InfoBlockForm id="infoBlockForm"/>
    </mx:ViewStack>

</mx:Canvas>
```

Oben sieht man die Implementierung der Editor View Klasse, dem Hauptcontainer der gesamten Applikation. Der Editor besteht aus dem `InfoBlockBrowser` und der `InfoBlockForm`. Beide Komponenten wurde wiederum in einen `ViewStack` platziert, damit zu einem Zeitpunkt nur eine der beiden angezeigt wird. Die Auswahl der angezeigten Komponente wird über das `ViewStack` Property `selectedChild` gesteuert. Der `ViewStack` kennt die Unterkomponenten anhand deren ID Properties. In das Feld `selectedChild` des `ViewStacks` wird also die ID der Komponente gesetzt, die angezeigt werden soll - `"infoBlockBrowser"` oder `"infoBlockForm"`.

Nach dem Presentation Model Pattern wird die Interface Logik zum Umschalten der `ViewStack` Seiten in die PM Klasse platziert. Wie man an dem Codesausschnitt sehen kann, wird diese Logik in die `EditorPM` Klasse ausgelagert. Das `EditorPM` wird durch den IoC Container Parsley injected. Hier wird ein `FastInject` MXML Tag hierfür verwendet, damit nicht die gesamte View Klasse von dem Container gemanaged werden muss.[7] Dieses enthält ebenfalls wie der ViewStack eine Zustandsvariable `selectedChild`. Nun wird ein Databindung von dem Property `selectedChild` im EditorPM zu dem Peroperty `selectedChild` im `ViewStack` konfiguriert. Wenn man nun die angezeigte Komponente bestimmen möchte, trägt man einen entsprechenden ID Bezeichner im `EditorPM.selectedChild` ein. Der `ViewStack` im `Editor` View wird automatisch durch das Databinding aktualisiert.

Die Entscheidung zum Wechsel der ViewStack Seite findet selbst entweder in dem `InfoBlockBrowser` oder in der `InfoBlockForm` statt. Der Autor wählt eine ID im `InfoBlockBrowser` aus, so wird auf die `InfoBlockForm` gewechselt oder der Autor drückt den Zurück-Button in der `InfoBlockForm` und gerät wieder zum `InfoBlockBrowser`. Die Logik für die Behandlung der Benutzereingabe wird in die entsprechenden PM der jeweiligen Display-Komponente ausgelagert. Dabei kennen sich die PM's unter einander nicht. Es ist aber eine Kommunikation zwischen den drei PM's erforderlich. Das `InfoblockBrowserPM` teilt dem `InfoBlockFormPM` die gewählte `InfoBlock` ID mit, worauf das PM den kompletten Infoblock lädt. Gleichzeitig muss das `Infoblock`-

7 Die Alternative wäre ein Actionscript [Inject] Metatag. Hierfür muss sich das Objekt, welches diese Metatags verwenden vom Parsley Container erzeugt werden, was für Views nicht möglich ist, da sie von dem Flex Framework selbst erzeugt werden müssen. Es gibt zwar eine Möglichkeit, eine graphsische View Klasse nachträglich zum Parsley hinzuzufügen, jedoch ist der MXML FastInject schneller und vollkommen ausreichend, da die Interaktionen in den PM's stattfinden.

`BrowserPM` dem `EditorPM` mitteilen, dass dieses nun dafür sorgen muss, dass die `InfoBlockForm` angezeigt wird, um den geladenen InfoBlock darzustellen. Diese Anforderung wird, wie bereits geschildert, durch das Messaging Feature von Parsley gelöst. Die Realisierung dieses Prozesses soll nun an Code Auschnitten verdeutlicht werden.

InfoBlockBrowser. mxml :

```
<mx:Canvas>

    <mx:Script><![CDATA[

        [Bindable]
        public var model: InfoBlockBrowserPM;

    ]]></mx:Script>

   <parsley:FastInject property="model" type="InfoBlockBrowserPM">

    <mx:Tree dataProvider="{model.infoBlocksGraph}" width="100%" height="100%"
        doubleClickEnabled="true"
        itemDoubleClick="model.editInfoBlock(event.currentTarget.selectedItem)">

    </mx:Tree>

</mx:Canvas>
```

Der `InfoBlockBrowser` besitzt entsprechend ein `InfoBlockBrowserPM` unter der Instanzvariable `model`. Das `InfoBlockBrowserPM` wird von Parsley injected. Der `InfoBlockBrowser` enthält weiterhin ein Tree-Widget, welches den InfoBlocks-Nachfolger-Graphen zeichnet. Das Tree-Widget reagiert auf ein Doppelklick auf einen seiner Items. Die Items sind die InfoBlock ID-Bezeichner. Wird so ein Doppelclick ausgeführt feuert das Tree-Widget das `ItemDoubleClickEvent` ab. Zu diesem Event wird in MXML ein Inline-Event-Listener registriert - `itemDoubleClick="model.editInfoBlock(...)"`. Dieser Listener ist eine Methode des `InfoBlocksBrowserPM`.

InfoBlockBrowserPM.as:

```
class InfoBlockBrowserPM extends EventDispatcher {
```

```
    [Event(name="infoBlockSelected",
           type="de.tk.editor.application.EditorNavigationEvent")]

    [ManagedEvents("infoBlockSelected")]

    [Bindable]
     public var infoBlocksGraph;

    [Inject]
     public var infoBlocksDelegate;

    public function editInfoBlock (infoBlockPreview: InfoBlockPreview):void {

        this.dispachtEvent (
            EditorNavigationEvent.newInfoBlockSelectedEvent( infoBlockPreview )
        );

     }

    public function getInfoBlocksGraph () :void {
        var resultHandler = function (event:ResultEvent){
            this.infoBlocksGraph = event.result;
        }
        var faultHandler = ...;
        InfoBlocksDelegate.getInfoBlockPreviews (resultHandler, faultHandler);

    }

}
```

Wenn der `editInfoBlock` Listener im `InfoBlocksBrowserPM` getriggert wird, so dispatcht dieser einfach ein neues Event mit dem selektiertem Element aus dem Tree-Widget. Dieses eigene Event bekommt den Datentyp `EditorNavigationEvent`.

EditorNavigationEvent.as :

```
class EditorNavigationEvent extends Event {
   private static const INFO_BLOCK_SELECTED:String = „infoBlockSelected" ;
```

```
private static const EDIT_COMPLETE:String = „editComplete“;

public var selectedInfoBlock;
public EditorNavigationEvent (type:String) {
   super(type, false, false);
}

public static newInfoBlockSelectedEvent
       (infoBlockPreview): EditorNavigationEvent
{
   var event =  new EditorNavigationEvent (INFO_BLOCK_SELECTED);
   event.selectedInfoBlock = infoBlockPreview;
   return event;
}

public static newEditCompleteEvent { ... }
```

Events in ActionScript können einen String `type` Property zugewiesen bekommen. Somit kann ein Event Klasse mehrere Events repräsentieren, wenn sie jeweils mit einem unterschiedlichen `type` Property instanziert wird. Das `EditorNavigationEvent` soll für beide Ereignisse - Auswählen eines `InfoBlocks` und Zurücknavigieren auf den Browser nach Editionsabschluss - verwendet werden. Um Typfehler in den `type` Strings zu vermeiden, wird diese Erzeugung des `EditorNavigationEvent` mit den beiden Typstrings „`infoBlockSelected`“ oder „`editComplete`“ in die statischen Methoden `EditorNavigationEvent.newInfoBlockSelectedEvent()` und `EditorNavigationEvent.newEditCompleteEvent()` ausgelagert.

Normalerweise, um auf ein abgefeuertes Event reagieren zu können, muss ein Eventlistener den Eventdispatcher direkt kennen. Das Messaging Feature von Parsley erlaubt es auf einen expliziten Aufbau dieser Beziehungen zu verzichten. Das Ganze wird wieder mit ActionScript Metatags konfiguriert. Zunächst wird, wie man aus dem `InfoBlockbrowser` Code sieht, das Event für Parsley mit einem `[Managed]` Tag annotiert. Nun muss in der Komponente, die auf dieses Event reagieren will, eine Methode mit einem `[MessageHandler]` Tag gekennzeichnet werden.

Auf das `EditorNavigationEvent` sollen nun sowohl das `EditorPM`, als auch das `InfoBlockFormPM` reagieren. Das `InfoBlockPM` lädt den gewählten `InfoBlock` und das `EditorPM` schaltet die Anzeige von dem `InfoBlockBrowser` auf die `InfoBlockForm`, um und den geladenen `InfoBlock` anzuzeigen.

InfoBlockPM.as :

```
class InfoBlockPM {

[Bindable]
public var infoBlock:infoBlock;

[Inject]
public var InfoBlockDelegate;

  [MessageHandler (selector="infoBlockSelected", order="2")]
  public function getInfoBlock (event: EditorNavigationEvent) :void {
      var resultHandler = function (event:ResultEvent){
          this.infoBlock = event.result;
      }
      var faultHandler = ...;
      InfoBlocksDelegate.getInfoBlock (
           event.selectedInfoBlock.id,
           resultHandler,
           faultHandler
       );

  }

   ...

}
```

Das `InfoBlockPM` reagiert also auf das Abfeuern des `EditorNavigationEvent` mit dem `type` String `"infoBlockSelected"` durch das `InfoBlockBrowserPM` und ruft die eigene Methode `getInfoBlock` auf. Diese ruft wiedereum vom Server den kompletten `InfoBlock` ab und speichert ihn in der `[Bindable]` Instanzvariable `infoBlock`, wodurch sich die diese Variable beobachtende `InfoBlockForm` aktualisieren kann. Damit der Aufruf sichergestellt wird, muss die Mehtode zunächst den richtigen Eventdatentyp als Parameter spezifizieren - `getInfoBlock (event: EditorNavigationEvent)`. Falls die Eventklasse nur ein Ereignis repräsentiert , reicht es anschließend die Methode mit dem Parsley `[MessageHandler]` Tag zu kennzeichnen. Da in

diesem Fall das Event verschiedene Typen besitzt, muss zusätzlich in dem Tag der richtige Typ - `selector="infoBlockSelected"` - angegeben werden. Ferner wird mit `order="2"` eine Priorität festgelegt, denn bei diesem Szenario reagieren zwei Listener auf das gleiche Event - `InfoBlockPM` und `EditorPM` und dieser Messagehandler soll zuerst ausgeführt werden.

Das EditorPM trägt bei dem gleichen Event lediglich den ID Bezeichner - „infoBlockForm" in das `[Bindable]` Property `selectedChild` ein.

EditorPM.as:

```
class EditorPM {
[Bindable]
public var selectedChild:String;

[MessageHandler (selector="infoBlockSelected", order="1")]
  public function showForm (event: EditorNavigationEvent) :void {
         this.selectedChild = "infoBlockForm";
  }
  ...
}
```

Auf die gleiche Art und Weise stellt man es auch her, dass die Benutzereingabebehandlung in dem `InfoBlockFormPM` zum Zurücksckschalten auf den `InfoBlockBrowser` an das `EditorPM` mitgeteilt wird. `InfoBlockFormPM` feuert das Parsley gemangete `EditorNavigationEvent` ab und das `EditorPM` muss eine weitere Methode für diese Behandlung kennzeichnen.

6.3. Pflege des InfoBlock Zustandes

Es wurde entschieden den InfoBlock Zustand über mehrere View Komponenten in dem Frontend der Applikation zu pflegen. Hierfür werden dynamisch graphische Elemente erzeugt und angezeigt. Dabei lautet die Vorgehensweise, die Erzeugung der Views für die Unterelemente des InfoBlocks abhängig von seinem Datenzustand zu machen. D.h. wenn der Autor z.B. eine neue Responce zum InfoBlock hinzufügen will, so wird zunächst ein neues Responce Objekt in das responces Array des Infoblocks hinzugefügt und per Databinding werden die `ElementPreivew` Komponenten zum Vorschau der Responces erstellt.

Diese dynamische Erzeugung der graphischen Klassen kann in Flex mit sog. ItemRenderer realisiert werden. Es gibt mehrere Widgets die solche ItemRenderer unterstützen. Eins davon ist

das List-Widget. Normalerweise repräsentiert das List Widget alle Einträge als Text. Setzt man es jedoch in Kombination mit einem ItemRenderer ein, so kann man erreichen, dass jeder Eintrag in der Liste durch eine eigene graphische Komponente gerendert wird. Es wird also, Beispielweise, ein List-Widget verwendet, um alle Responces des InfoBlocks anzuzeigen. Die Responce selbst soll mit einen ItemRenderer durch eine eigene graphische Klasse visualisiert werden. An diesem Beispeil der Responce Verwaltung soll die Pflege des InfoBlocks in der Appliaktion und die dynamische Erzeugung der Pflegemasken deutlich werden.

InfoBlockForm.mxml :

```
<mx:Canvas>

      <mx:Script><![CDATA[

        [Bindable]
         public var model: InfoBlockFormPM;

    ]]></mx:Script>
   <parsley:FastInject property="model" type="{InfoBlockFormPM}"/>

   <presentation:ResponcesList data={model.infoBlock.responces}/>

    <!-- Preamble -->
        ...
    <!-- Info -->
        ...
    <!-- Challenge -->
        ...

</mx:Canvas>
```

Zunächst wird die Darstellung der Responces in eine eigene graphische MXML Komponente `ReponcesList` ausgelagert. Diese soll für die CRUD Operationen der Responces verantworlich werden. Die `ResponcesList` wird dafür per Databinding mit den InfoBlock `responces` Property verbunden - `ResponcesList data={model.infoBlock.responces}`. Die `ResponcesList` bekommt nun alle Änderungen an den Responces des InfoBlocks in der InfoBlockForm mit. Nun soll die `ResponcesList` Implementierung betrachtet werden.

ResponcesList.mxml :

```
<mx:Canvas>

      <mx:Script><![CDATA[

       [Bindable]
        public var model: ResponceListPM;

       override public function set data (responces:ArrayCollection){
           model.responces = responces;
       }

    ]]></mx:Script>
   <parsley:FastInject property="model" type="{ResponcesPM}"/>

   <!--Button für neue Responce Anlegen -->
    <Mx:Button id="new" label="+"
        click="model.addNewResponce()"
    />

 <!-- Responces Anzeigen -->
   <mx:List
      dataProvider="{model.responces}" >
      itemRenderer="ElementPreview"
    />
</mx:Canvas>
```

Von Außen sah es so aus, als ob die `ResponcesList` ein Property, also eine Zustandsvariable `data` besäße, was man an dem Aufruf der Komponente in der `InfoBlockForm` feststellen konnte - `ResponcesList data={model.infoBlock.responces}`. Die Verwaltung der Responces soll natürlich wiederum ein Presentation Model - `ResponcesListPM` - übernehmen. Wenn also die `ResponceList` durch Databinding auf die Änderungen im InfoBlock reagiert, soll der Zustand in sein PM wandern. Hierfür wird das von Außen als Property erkennbare Attribut `data`, intern als eine Setter-Funktion deklariert - `public function set data`. Diese Funktion trägt eine Referenz zu den InfoBlock Responces in die `ResponcesListPM` Zustandsvariable `responces` ein. Somit wird

sichergestellt, dass das `ResponcesListPM` auf dem Zustand des InfoBlocks arbeiten kann und zwar nur an dem für diese Klasse zuständigen Bereich - den Responces.
Die `ResponcesList` zeigt nun die Responces durch das List-Widget. Damit das Widget immer die aktuellen Responces anzeigt, wird der Dataprovider des List-Widgets mit den Daten im `ResponceListPM` verbunden - `List dataProvider="{model.responces}"`. Zusäztlich wird hier ein Itermrenderer definiert, damit jede Responce durch eine eigene View Klasse - `ElementPreview` - in der Liste auftaucht. Wenn also das ResponcesListPM responces einfügt oder entfernt, so aktualisiert sich das List-Widget automatisch. Dabei findet eine Iteration über alle Responces statt und es wird für jede eine `ElementPreview` Klasse erzeugt und auf dem Display dargestellt. Dabei wird die jeweilige Responce automatisch an das `data` Property des Itemrenderers übergeben. Das `ElementPreview` deklariert wiederum das `data` Property als Setter und leitet die Referenz an sein PM weiter.

ElementPreivew.mxml:

```
<mx:Canvas>
 <!-- Vorschau für diveres InfoBlockUnterlemente: Responce, Preabmle, Info u.a  -->
      <mx:Script><![CDATA[

       [Bindable]
        public var model: ElementPM;


       override public function set data (element){
           model.element = element;
       }

    ]]></mx:Script>
   <parsley:FastInject property="model" type="{ElementPM}"/>

 <!-- DropDown zur Auswahl der weiteren Editoren e.g. UpdateEffectsEditor -->
    <mx:PopUpMenuButton  ...
       click="model.openEditor(event)  />

<!-- Zeige den Text des Elements -->
 <mx:Textarea text = {model.element.text}    />

</mx:Canvas>
```

Das `ElementPM` wird natürlich weitere Masken aufrufen und die Unterelemente seines Elements auf die gleiche Art und Weise an die jeweiligen Masken übergeben wie oben gezeigten Komponenten. Z.B. wenn der Benutzer weiterhin das Preview einer Responce auswählt und im Drop-Down-Menü den `UpdateEffectsEditor` aufruft, um UpdateEffects an der selektierter Responce zu annotieren.

6.4. Reasoner Formel Notationen

Die boolischen Verknüpfungsformeln in den DeactivateEffects oder Conditions werden alle in einer Prefix Notation defeniert. Diese Notation ist für Menschen im Gegensatz zu der üblichen-Infix Notation schwieriger zu lesen. Bei den komplexen boolischen Formeln in Conditions und DeactivateEffects wirken diese Formeln sehr verwirrend, und ein manuelles parsen erfordert einen hohen Aufwand. Deswegen wurde an dieser Stelle ein Parsergenerator namens ANTLR eingesetzt, der aus einer Grammatik, einen Prefix und einen Infix Parser in ActionScript erzeugt.

Beispiel :

Die DeactivateEffect Formel in Prefix Notation

```
 OR (
       AND ((Benutzer Wissenschaft interessiert)( Interesse groß))
      (Benutzer Religion interessiert)
)
```

soll in die verständliche Infix Notation für die Anzeige übersetzt werden

```
(Benutzer Wisssenschaft interessiert) AND (Interesse groß))
 OR  (Benutzer Religion interessiert)
```

Zu besseren Übersicht wird anschleißend die Formel in die Bestandteile zerteilt, diese werden in eine Tabelle platziert und die Formelbestandteile durch Tabellenindexes ersetzt.

1 `Benutzer Wisssenschaft interessiert`

2 `Interesse groß`

3 `Benutzer Religion interessiert`

Formel : `($1 AND $2) OR $3`

Dabei wird folgendermaßen verfahren. Der generierte Parser erstellt entsprechend der Grammatik einen Syntaxbaum mit boolischen Operatoren als Knoten und Expressions als Blätter.

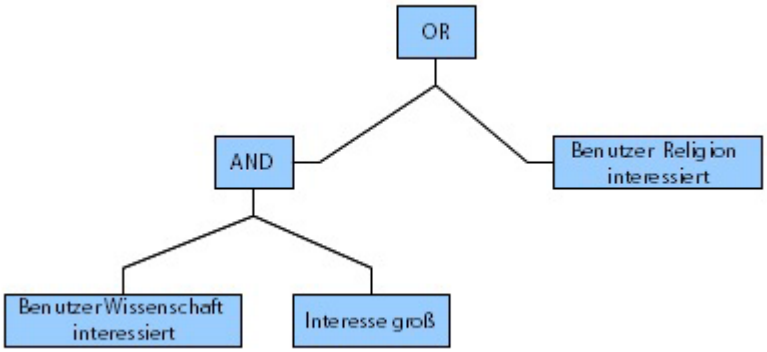


Abbildung 6.1: Syntaxbaum beim Lesen

Der Syntaxbaum wird rekursiv in der Infix Order durchlaufen und als String ausgegeben, wobei wenn immer an einem Blatt angekommen, ein Expression Objekt erzeugt in ein Array gepackt wird und anstatt des Objektes selbst sein Arrayindex ausgegeben. Die graphische Tabelle bindet einfach an das Array und unten wird der geparste String `($1 AND $2) OR $3` in einem Textfeld angezeigt.

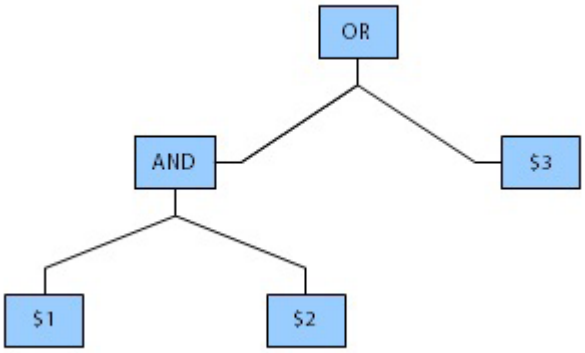


Abbildung 6.2: Syntaxbaum beim Schreiben

Bei der Eingabe der Formel verfährt man genau umgekehrt, so dass zunächst folgender Syntaxbaum aus der eingegbenen Formel entsteht.

6.5. Nicht oder teilweise realisierte Funktionen

Dieser Abschnitt geht auf die nicht realisierten, aber wichtigen Funktionen ein.

6.5.1. InfoBlock Nachfolger CRUD

Die InfoBlock Referenzen, für die fest verpointernten InfoBlocks wurden zwar angezeigt, jedoch wurde nicht die Möglichkeit implementiert neue Nachfolger zu einem InfoBlock hinzufügen. Die Funktion an sich ist trivial. Es müssen lediglich die ID's der Nachfolger-InfoBlocks in dem Eltern-InfoBlock gespeichert werden. Die Schwierigkeit ergibt sich aus der Benutzbarkeit. Den einem Autor ist es nicht zuzumuten sich stets die richtigen ID's merken zu müssen. Wünschenswert wäre ein Drag und Drop oder Feature mit dem die ID's aus dem InfoBlockBrowser in ein Bereich der InfoBlockForm per Drag und Drop übertragen werden. Hierfür wäre aber ein befinden beider Komponenten gleichzeitig auf dem Bildschirm notwendig, worauf Momentan aus Platzgründen verzichtet wird. Dei Funkton müsste also wie bie den anderen Editoren über Fenster Popups gelöst werden, die minimiert und maximiert werden können. Denkbar wäre auch Kontexteditor im InfoBlockBrowser selbst.

6.5.2. Richt Text und Block Annotierung

Der mit Style Tags markierte Text wird mittels XSLT in einer von dem Flash Player verständliche Form übersetzt. Wenn man nun Attribute an dem HTLM Markup beibehält, die diese Elemente nicht verstehen, so werden sie von den Textwidgets einfach ignoriert und verworfen.

Beispiel eines Condref getaggten Blocktextes:
<para condref="interessierter"> Sie interessieren sich also sehr dafür </para>

Dieses <para> Tag wird zu HTML Tag übersetzt, so dass das Textwidget ein Absatz anzeigt.
 Sie interessieren sich also sehr dafür

Das Textwidget, zeigt zwar diesen Text an,obwohl condref kein richtiges HTML Attribut ist, jedoch, wenn es nach seinem HTML Text befragt wird das Attribut verworfen. Somit wenn der Autor den Text bearbeitet, kann hinterher nicht mehr festgestellt werden, welches Abschnitt mit einer Condref getagged war.

Dieses könnte umgangen werden indem aus dem aus dem condref Attribut, ganz einfach ein valides HTML Attribut, wei id oder class gemacht wird. Dennoch wird eine Inplace Annotierung schwierig, da aus der Cursor Position im Textwidget nicht auf die Position im HTML Markup geschlossen werden kann.

6.5.3. Dynamische Annotation Completions

Die Analyse zeigte, dass der Ontolgie Reasoner Racer Ontolgie Formelingaben erlaubt, ohne dass zuvor Vorbedingungen in der TBox geschaffen werden müssen. Momentan wird aber nur die TBox für die Vervollständigungen durchsucht . Um solche Eingaben mit Textvervollständigung zu unterstützen können müssten alle InfoBlock Annotationen durchsucht werden, da diese benutzerspezifisch sind und nicht in der TBox gespeichert werden können. Hierfür ist aber ein Umstellung des Dialog Persistenzmechanismus erforderlich.

7. Zusammenfassung

Um die Arbeit zusammenzufassen, wird auf den aktuellen Stand der Entwicklung eingegangen und ein Ausblick gegeben. Abschließend erfolgt eine Bewertung der Arbeit.

7.1. Stand der Entwicklung

Bis zum Abschluss dieser Arbeit wurden die Hauptanforderungen aus Analyse 3.4 ansatzweise gelöst. Einige Unteranforderungen wurden wegen eines angemessenen Zeitrahmens nur theoretisch behandelt.

Die Erstellung der InfoBlock Nachfolger-Beziehungen ist erfühlt. Diese werden durch ein Tree-Widget dargestellt. Das Einfügen neuer Beziehungen wurde theoretisch behandelt. Der Autor kann ein InfoBlock auswählen und bekommt eine Vorschau. Es ist möglich die InfoBlock Bestandteile - Preambel, Info, Challlenge und Responces zu bearbeiten. Ferner ist es möglich an diese Elemente die benötigten Annotation anzuhängen. Zunächst sind es die Masken zu Bearbeitung von UpdtateEffects, Deactivate Effects der Responces und von Facts des InfoBlocks. Diese Masken erlauben eine Eingabe mit textueller Autovervollständigung der Bezeichner aus der TBox des Dialogsystems. Eine mögliche Durchsuchung aller bisher angegeben Annotationen für die Autovervollständigung wurde diskutiert, jedoch nicht umgesetzt. Ebenfalls das Anhängen der Condition-Abfragen ist möglich. Die Conditions werden analog zu dem Systemformat in einer separaten Tabelle verwaltet. Sie können mit einer Formeleingabe Maske erstellt werden. Für die boolisch-verknüpften Ausdrücke aus der DeactivateEffects Annotationen und Conditions wurde ein Parser zu Übersetzung von der schlecht lesbaren Postfix Notation zu der gewohnten Infix Notation implementiert. Für die Anforderung [7.Richtext] wurden exemplarisch einige Style-Tags mittels XSLT in eine HTML Form übersetzt, so dass dieser HTML gestylte Text, durch die Flex Text-Widgets graphisch formatiert wird. Eine editorielle Unterstützung bei Condref Annotation der einzelnen Textblöcke konnte wegen überproportionalem Aufwand nicht umgesetzt werden. Es is jedoch eine Texteingabe ohne Vervollständigung möglich.

7.2. Ausblick

Die Editorapplikation bringt die grundlegenden Tools zu Bearbeitung des Systems. Diese könnten durch folgende Aspekte benutzerfreundlicher gestaltet werden:

1. Ermöglichen einer Konfliktauflösung beim parallelem Arbeiten mehrere Benutzer.
   Hierfür sollte man bevorzugt das Persistenzdatenmodel zu einer Datenbank ändern. Die Änderung der einzelnen InfoBlock Elemente sollte sofort gespeichert werden. Denkbar sind

Sperrern der Elemente, falls diese sich in der Bearbeitung anderer Benutzer befinden oder einfache Benachrichtigungen, falls ein Element zwischenzeitlich geändert wurde.

2. Durchsuchen und Refactoring
   Die InfoBlock Inhlate sollten durchsuchbar sein um bei einer großen Datenmenge den Überblick behalten zu können. Es wäre denkbar bestimmte Bezeichner in den Annotationsausdrücken durchgehend zu ändern wollen.

3. Bessere Nachfolger Visualisierung.
   Hier könnte die Darstellung mittels einer Graphenvisualisierung überschauberer werden.

4. Bessere Ontologie Unerstützung
   Es gibt erfolgreiche Ansätze um Ontologien zu visualisieren und somit die Begriffe und Beziehungen aus dem der Maschine bekannten Weltausschnitts anschaulich zu machen. Momentan könnte ein solches Tool verwendet zur Anschau verwendet, werden da die Ontologie in dem standardisierten OWL Format vorliegt., die gefunden Bezeichner müssten aber händisch eingetippt werden, was mit Autovervollständigung natürlich einfach ist, aber allein für sich eben keine Übersicht über alle Begriffe schafft. Eine Intergration der Visualisierungsmethoden von Ontologien aus bekannten Tools, würde eventuell zum besseren Benutzererlebnis beitragen.

5. WYSIWYG Modus
   Die Autoingsofware wählt einen Datenzentrierten Weg für die Bearbeitung der Inhalte. Von dem Benutzer wird eine Transferleistung von dieser Datenansicht zu der tatsächlichen Ansicht erwartet. Denkbar wäre ein Modus, bei dem das Dialogsystem von dem Administrator aus der Benutzersicht durchlaufen wird und währenddessen die Textinhalte live annotiert werden können.

7.3. Bewertung der Arbeit

Die Arbeit beschäftigte sich primär mit der Aufgabe ein desktopähnliches Administrator Interface als ein Teil der Webanwendung zu entwerfen. Architektonisch wird ein Weg gezeigt, wie in Flex nicht nur Trennung von Layout und Information, sondern auch eine Auslagerung der Interface Logik, für bessere Testbarkeit und Wiederverwendung, durch das Presentation Model Pattern erreicht werden kann. Außerdem wurde gezeigt wie durch Dependcy Injection und eventbasiertes Messaging die einzelnen Komponenten in einer Anwendung auf ein lose gekopplete Weise kommunizieren können. Die Analyse des Dialgosystems selbst liefert eine Abgrenzung von der theoretisch komplexen Ontologie Lehre und erklärt es an einem pragmatischen Beispiel

für technisch nicht affine Personen.

Kritisch zu betrachten ist, ob der gewonnene Wert aus der Autoringsoftware gemessen an dem dafür benötigten Aufwand angemessen ist. Hierfür sollten die Meinung von potentiellen Benutzern untersucht werden und geprüft werden ob die Editorsoftware ein besseres Verständnis des Dialogsystems überhaupt schaffen kann. Außerdem wird die Funktionsweise des Systems selbst nicht hinterfragt.

Abbildungsverzeichnis

Literaturverzeichnis

[JJGar05] `Jesse James Garrett`, **Ajax: A New Approach to Web Applications**, *http://adaptivepath.com/ideas/essays/archives/000385.php*, Stand: Februar 2005

[Adobe3D10] `Adobe Systems Inc`, **Animation Learning Guide for Flash CS4 Professional**, *http://www.adobe.com/devnet/flash/3d_animation.html,* Stand: April 2010

[All02] `Allaire Jeremy`, **Macromedia Flash – A next Generation Rich Client**, 2002

[AOS10] `Adobe Open Source`, **Cairgorm Guidelines**, *http://opensource.adobe.com/wiki/display/cairngorm/CairngormGuidelines,* 21 Mai 2010

[Blaze10] `Adobe Systems Inc`, **Explicitly mapping ActionScript and Java objects**, *http://livedocs.adobe.com/blazeds/1/blazeds_devguide/help.html?content=serialize_data_3.html*, 2010

[Crock10] `Douglas Crockford`, **Crockford on JavaScript -- Episode IV: The Metamorphosis of Ajax**, *http://developer.yahoo.com/yui/theater/video.php?v=crockonjs-4*, März 2010

[DeWikiWeb10] *http://de.wikipedia.org/wiki/Webanwendung*, Stand Mai 2010

[EnWikiWeb10] **http://en.wikipedia.org/wiki/Web_application**, Stand Mai 2010

Flex08] `Adobe Systems Inc`, **Adobe Flex 3 Developers Guide**, 2008

Flex10] `Adobe Systems Inc`, **Adobe Flex,** *http://www.adobe.com/products/flex/*, Stand 2010

[Fow02] `Martin Fowler`, **Patterns of Enterprise Application Architecture**, Addison Wesley, 15 November 2002

[Fow06] `Martin Fowler`, **Development of Further Patterns of Enterprise Application Architecture,** 2006, http://martinfowler.com/eaaDev/, Stand: Mai 2010

[GWT10] `Google Inc`, **Google Web Toolkit,** http://code.google.com/intl/de-DE/webtoolkit/ , Stand: 2010

[GWTRPC10] Google Inc, **Google Web Toolkit - Making Remote Procedure Calls**, http://code.google.com/intl/de-DE/webtoolkit/doc/latest/tutorial/RPC.html, Stand: April 2010

[Halder04] Alexander Halder, **Entwurf und Nutzung von Ontologien zur Produktteileverwaltung am Beispiel des Geschäftsfeldes PKW der DaimlerChrysler AG**, Ulm, 2004

[Hev08] Miško Hevery, **Clean Code Talks – Dependency Injections**, http://misko.hevery.com/2008/11/11/clean-code-talks-dependency-injection/, Google Tech Talks , November 2008

[Jan08] Gerhard Janisch, **Analyse von Rich Internet Application Frameworks am Beispiel einer Thesaurusverwaltung**, AUT Hagenberg, Juni 2008

[Java10] Sun Microsystems, **Enhance User Experiences with JavaFX 1.3 software**, http://javafx.com, Stand 2010

[Liu09] Jeff Liu, **ZK vs. GWT,** *http://www.zkoss.org/smalltalks/gwtZk/#scAndcc*, März 2009

[LR06] Bernhard Lahres, Gregor Raýman, **Praxisbuch Objektorientierung,** Galileo Computing, 2006, *http://openbook.galileocomputing.de/oo/index.htm,* Stand: Mai 2010

[Mar08] Sebastian Martens, **Technologievergleich von RIA Plattformen**, Eddelak 08, September 2008

[Rein08] Daniel Reinhart, **Testting Visual Componets with FlexUnit**, *http://cookbooks.adobe.com/post_Testing_visual_components_with_FlexUnit-7222.html*, Januar 2008

[SDN10] Sun Developers Network, **Core J2EE Patterns - Data Access Object**, *http://java.sun.com/blueprints/corej2eepatterns/Patterns/DataAccessObject.html*, Stand Mai 2010

[Silver10] Microsoft Corporation, **The Official Microsoft Silverlight Site**, **http://www.silverlight.net**/, Stand 2010

[SilvLearn10] Microsoft Corporation, **Silverlight Learning Center**, *http://www.microsoft.com/germany/net/silverlight/learn.aspx*, Stand: 2010

[Sug09] Tom Sugden, **Applying the Presentation Model in Flex**, *http://blogs.adobe.com/tomsugden/2009/08/applying_the_presentation_mode.html*, August 2009

[WebLeo08] Steven Webster, Leon Tanner, **Developing Flex RIAs with Cairngorm micro-architecture**, *http://www.adobe.com/devnet/flex/articles/cairngorm_pt1.html* , 23 Mai 2008

[Will07] Paul Williams, **Presentation Patterns Introduction**, *http://blogs.adobe.com/paulw/archives/2007/09/presentation_pa.html*, Juli 2007

[ZK10] *Potix Corporation*, **ZK-Direct RIA**, *http://www.zkoss.org/* Stand: 2010

[ZKGuide10] Potix Corporation, **ZK – Developers Guide**, v.3.6.4, 2010

Glossar

Abox Benutzerspezifische Ontologie Wissensdatenbank

Ajax Asynchronious Javascript and XML. Ansammlung von Techniken um RIA's mit W3C Webstandards zu erzeugen.

API Applciation Programming Interface – eine Programmschnittstelle.

Backend hier: Der serverseitige Teil der Webanwendung

BlazeDS Middleware für die Kommunikation zwischen Flex Clients und Java Backends.

Cairngrom 2 MVC Microarchitektur zu Umsetzung des MVC Muster

Cairngrom 3 Blaupause zu Umsetzung des Presentation Model Musters

Cient Eine Maschiene welche über das Netzwerk Kontakt zu einem Server aufnimmt und seine Dienstleistung nutzt. Oft nur ein Programm auf der Maschine gemeint

DataBinding Eine Methode bei der Daten von einer Quelle in ein Ziel, bei Änderung automatissch Übertragen werden.

DI Dependency Injection – Ein Verfahren bei dem die Erzeugung der Objekt Bezeigungen an eine exteren Komponente übergeben wird

DOM Document Object Model - Abbildungs Model eines Dokuments im Arbeitspeicher. Meist verbunden mit einer Schnittstelle zum DOM Zugriff.

EventBubbling Evntpropagation die DisplayList hoch.

EventPropagation Bewegen eines Events entlang der DisplayList, so dass andere Komponenten daruf reagieren können

Flash Player DisplayList Eine Baumstruktur alle Graphischen Elemente in Flash. Analog zu Browser DOM, jedoch mit deutlich einfacher Zugrffsmöglichkeit.

Frontend hier: Der clientseitige Teil der Webanwendung

GWT Google Web Toolkit. Ein clientzentriertes Ajax Komponenten Framework.

InlineEventListener Eine Funktion, welche als MXML Attribut an ein MXML Property angehängt wird und im Fall eines bestimmten Events aufgerufen wird.

IoC Inversion of Control – Ein Verfahren bei dem die Komponente ihre Funktionen an ein Framework abgibt und diese bei einem späteren Zeitpunkt aufgerufen werden

JSON Java Script Object Notattion. Leichtgewichtiges stringbasiertes Dokumentenformat.

JVM Java Virtual Machine

Middleware Anwendungsneutrales Programm, dass zwischem Anwendungen vermittelt.

MVC Model View Controller – Ein Muster zu Trennung von Presäntation, Interaktion und Daten

MXML Macromedia Extended Markup Language – Erweiterte XML Sprache zu deklarativen Entwicklung von Benutzeroberflächen im Flex Framework.

MXML Custom Component Eigene graphische Komponente, die sich aus den Benutzeroberflächenelemente des Flex Frameworks zusammensetzt

Objekt Serialisierung Überführen eines Objekts in ein Datenformat, dass auf der Festplatte abgelegt werden oder übers Netzwerk übertragen werden kann.

Ontologie Wissenrepräsentationform für Maschienen

PM Presentation Model - Eine logische Abbildung der graphischen Klasse

Reasoner Software Einheit, welche auf der Ontologie Datennak Operiert – kann Abfragen formulieren oder Daten eintragen.

RIA Rich Intern Application. Eine Webanwendung, die auf der Clientseite ein desktopangelehntes Verhalten aufweist

Semantic Web Ein Pardigma, bei dem Inhalte im Internet für Computer verwertbar gemacht werden.
Server Eine Maschiene, welche eine Dienstleistung über das Netzwerk anbietet

Singelton Pattern Ein Muster bei dem eine und die selbe Instanz in einer statischen Variable gespechert und über die Klassenreferenz global verfügbar gemacht wird

SWF Shockwave Flash - Vom Flash Player abspielbares Programmarchiv.

Tbox Allgemeine Ontologie Wissensdatenbank

W3C World Wide Web Consortium (http://www.w3.org)

XML Extensible Markup Language – Auszeichnungssprache für Dokumenten.

Xpath XML Abfrage Sprache

XSLT Deklarative XML Transformation Sprache

ZK Zero Kelvin. Ein serverzentriertes Ajax Kompoenten Framework.