Hochschule für Angewandte Wissenschaften Hamburg

Hamburg University of Applied Sciences

Masterarbeit

Christian Stachow

Spezifikation und Entwicklung einer webbasierten Widget Orchestrierung für End-User Development

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Christian Stachow

Spezifikation und Entwicklung einer webbasierten Widget Orchestrierung für End-User Development

Masterarbeit eingereicht im Rahmen der Masterprüfung
im Studiengang Master Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Ing. Birgit Wendholt
Zweitgutachter: Prof. Dr. Stefan Sarstedt

Abgegeben am 29. August 2010

Christian Stachow

Thema der Masterarbeit

Spezifikation und Entwicklung einer webbasierten Widget Orchestrierung für End-User Development

Stichworte

End-User Development, Rapid Application Development, Rich Internet Application, Orchestrierung

Kurzzusammenfassung

In dieser Arbeit wird eine XML basierte Beschreibungssprache zur Orchestrierung von Widgets entworfen, die End-User mit geringer Erfahrung in der Web-Entwicklung erlaubt komplexe Web-Anwendungen durch die Verknüpfung von frei verfügbaren Widgets als Bausteine zu erstellen. Dazu werden typische Realisierungskonzepte des End-User Developments, relevante Standards und Produkte betrachtet und analysiert und die Anforderungen der Widget Orchestrierung bestimmt. Aufbauend auf den gewonnenen Erkenntnissen wird ein System entworfen und in einer prototypischen Implementierung erfolgreich auf seine Realisierbarkeit geprüft.

Title of the paper

Specification and development of a web orientated Widget orchestration for End-User Development

Keywords

End-User Development, Rapid Application Development, Rich Internet Application, Orchestration

Abstract

In this paper will be a XML based language for the orchestration of Widgets developed, which allows end-users with minimal experience in web-development to built complex web application with freely available widgets as building blocks. To achieve this, an analysis of typical concepts for implementation of end-user development, relevant standards and products will be done, to define the requirements for widget orchestration. With the use of the gained insight, a system will be developed and prototypical implemented to test the practicability

Inhaltsverzeichnis

1 Einleitung **8**
- 1.1 Motivation . . . 8
- 1.2 Zielsetzung . . . 9
- 1.3 Gliederung . . . 9

2 Vergleichbare Arbeiten **11**
- 2.1 End User Development . . . 11
  - 2.1.1 Konzepte . . . 13
- 2.2 Mashup . . . 18
  - 2.2.1 Widget . . . 22
  - 2.2.2 Mashlet . . . 23
  - 2.2.3 Orchestrierung . . . 24
  - 2.2.4 Produkte . . . 25

3 Analyse **33**
- 3.1 Anwendungsszenario . . . 33
  - 3.1.1 Organizer - Zeitplaner . . . 33
- 3.2 Anforderungen . . . 35
  - 3.2.1 Nichtfunktionale Anforderungen . . . 35
  - 3.2.2 Funktionale Anforderungen . . . 36
- 3.3 Technische Randbedingungen . . . 37
  - 3.3.1 Wiederverwendbare Widgets . . . 37
- 3.4 Mashlet . . . 39
  - 3.4.1 Widget Definition . . . 40
  - 3.4.2 Orchestrierung . . . 42
- 3.5 Zusammenfassung . . . 43

4 Design / Konzept **45**
- 4.1 Architektur . . . 45
  - 4.1.1 Kontext-Daten . . . 46
  - 4.1.2 Widget . . . 49
  - 4.1.3 Mashlet . . . 52
  - 4.1.4 Inter-Widget-Kommunikation . . . 53

4.1.5 Wiring 53
4.2 Abschluss 56

5 Realisierung **57**
5.1 Widget 57
5.2 Mashlet 60
5.3 Laufzeitumgebung 63
5.4 Wiring Guidelines 64

6 Fazit und Ausblick **72**
6.1 Fazit 72
6.2 Ausblick 73

Literaturverzeichnis **74**

A Inhalt der CD-ROM **78**

Abbildungsverzeichnis

2.1 Alice WhyLine 12
2.2 Barista: Eine mit Medien angereicherte Annotation einer Java-Methode 13
2.3 LabVIEW Datenfluss und Bedienelemente 15
2.4 Croquet Quelltext Editierung 16
2.5 Mashup Struktur 19
2.6 Beispiel Widget Interaktion: Währungsrechner 20
2.7 Aufbau eines Mashups 21
2.8 Orchester: Mescheder Windband, 2008 [1] 25
2.9 IBM Mashup Center: Wiring 30
2.10 JackBe Presto Wires (Quelle: JackBe) 30

3.1 Zeitplaner Quelle: M. Minderhoud 33
3.2 Trivial Organizer aus einzelnen Mashlet-Komponenten 35

4.1 SOA Beziehungsstruktur: Dreieckmodell 45
4.2 Zugriff auf die Kontext-Daten 46
4.3 Komponenten eines Widgets 50
4.4 Sequenz Diagramm: Verarbeitungsschritte 55

Tabellenverzeichnis

3.1 Erfüllung der Anforderungen der Widget Definition von OpenAjax Metadata 1.0 Specification Widget Metadata 41
3.2 Erfüllung der Anforderungen der Widget Definition von W3C Widget 42
3.3 Erfüllung der Anforderungen der Widget Definition von OpenSocial Gadget . . 42

1 Einleitung

Lange Zeit galt die Softwareentwicklung als schwierig und aufwendig und war ausschließlich professionellen Softwareentwicklern vorbehalten. Die fortlaufende Weiterentwicklung von Technologien, Anwendungen und Techniken bieten zahlreiche neue Möglichkeiten den Softwareentwicklungsprozess zu vereinfachen und dadurch Nicht-Programmierern bzw. End-User die Möglichkeit zu bieten, anspruchsvollere Anwendungen und Lösungsansätze zu erstellen, das unter dem Forschungsgebiet des End-User Development steht.

Dank der technologischen Fortschritte entstanden unzählige Werkzeuge, darunter visuelle Editoren wie z.B. das von Lego MINDSTORM (LEGO Mindstorm) oder OpenMusic (OpenMusic), Mashup[2]-Editoren wie z.B. das von Yahoo-Pipes und viele weitere.

Der immer einfacher werdende Umgang dieser Werkzeuge und die gute Bedienbarkeit der erstellten Anwendung erhöht die Verbreitung und Akzeptanz dieser. Aufgrund der hohen Qualität und Zuverlässigkeit der Anwendungen, werden die Werkzeuge auch im professionellen Bereich eingesetzt, um dort Zeit und Kosten zu sparen.

1.1 Motivation

End-User Development findet in verschiedenen Bereichen statt, sei es in der Komposition neuer Musikstücke mittels OpenMusic oder der Erstellung von Web-Anwendung als Mashup. Besonders interessant ist die Mashup-Entwicklung, die sich mit dem Web 2.0 Hype etabliert hat. Mashups greifen bestehende Fremdinhalte auf, verwerten diese und erstellen neuen Inhalt, die auf nahezu jedem Webbrowser[3] lauffähig sind und können einfache Spielereien[4] oder auch alltagstaugliche Anwendungen wie themenorientierte Nachrichten beinhalten. Das Statistisches Bundesamt Deutschland stellte am 03.12.2009 fest, dass 73% der privaten Haushalte einen Internetzugang besitzen und dank immer einfacher werdender Werkzeuge, auch damit potentielle Mashup-Entwickler von alltagstauglichen Anwendungen sind. Die aktuellen Mashup-Produkte sehen Laien ohne jegliche Erfahrung in der Programmierung bis hin zu erfahrenen

[2]Mischen von verschiedenen Inhalten

[3]Webbrowser (oder allgemein auch Browser genannt) sind spezielle Computerprogramme zum Betrachten von Webseiten im World Wide Web (Wikipedia).

[4]Neugierige Personen, die etwas Neues ohne ernste Absichten erstellen

Entwicklern von Web-Anwendungen als ihre Zielgruppe an. Je erfahrener die Zielgruppe desto komplizierter und flexibler ist das Mashup-Produkt. Das Konzept des Mashups beschränkt sich jedoch bei den meisten Produkten auf die Programmierung der Datenverarbeitung. Die Programmierung der Daten-Präsentation fehlt entweder vollständig oder sind als parametrisierbare Templates realisiert. Einige wenige erlauben auch die Erstellung der Daten-Präsentation über Widget Komponenten, die frei platziert und miteinander interagieren können. Solche erstellten Daten-Präsentationen lassen sich nur auf dem jeweiligen Mashup-Produkt ausführen. Diese Einschränkung betrifft zum gewissen Grad auch die programmierte Datenverarbeitung eines Mashups. Die meisten Mashup-Produkte nutzen Eigenlösungen zur Programmierung der Datenverarbeitung und eine Unterstützung von offenen Standards wie EMML ist nicht weit verbreitet. Die Anwendung des Mashups Konzepts auf die Daten-Präsentation, erlaubt die Erstellung komplexer Benutzerschnittstellen selbst durch End-User mit wenig Erfahrung.

1.2 Zielsetzung

Der Fokus dieser Arbeit liegt in der Bereitstellung einer Spezifikation einer Sprache, die die Verknüpfung von GUI-Elementen in Form von Widgets ermöglicht, um benutzerdefinierte interagierende Verhaltensmuster zu definieren. Die GUI-Elemente werden als Bausteine betrachtet, die verwendet werden um eine Anwendungslogik zu realisieren.

Die Komplexität der Sprache soll dabei so gering gehalten werden, dass selbst Programmierer bzw. End-User mit geringer Erfahrung in Web-Entwicklung in der Lage sein werden, ohne großen Lernaufwand zügig eigene Software-Konstrukte zu erstellen. Für eine vollständige Spezifikation muss auch das Layout der GUI-Elemente berücksichtigt werden, welches aber in dieser Arbeit nicht behandelt wird.

Der Beweis der Machbarkeit wird über einen Prototyp erbracht, der nur wichtige Aspekte der Spezifikation für die Realisierung eines Anwendungsfalls implementiert.

1.3 Gliederung

Die Arbeit teilt sich in fünf große Abschnitte auf:

Im Kapitel 2 werden die für diese Arbeit relevanten Arbeiten beschrieben und eine Vorstellung des Themenbereichs geliefert.

Im Kapitel 3 beschäftigt sich mit der Analyse der gestellten Aufgabe. Es präsentiert die relevanten Spezifikationen und Standards und legt die Anforderungen an das zu entwickelnde System fest.

Im Kapitel 4 wird das Design des zu entwickelnden Systems vorgestellt. Dazu werden die einzelnen Komponenten, die notwendigen konzeptuellen Standards und deren Zusammenhang erörtert.

Die für die Realisierung relevanten Entscheidungen werden in Kapitel 5 dargestellt. Dort werden die wesentlichen Implementierungsdetails beschrieben und an XML und XML-Schema Auszügen verdeutlicht.

Zum Schluss werden in Kapitel 6 die erreichten Ziele zusammengefasst und ein Ausblick gegeben.

2 Vergleichbare Arbeiten

Ziel dieser Arbeit ist die Entwicklung einer einfach zu programmierenden Anwendung zur Koordinierung und Steuerung von einzelnen Webbrowser-Anwendungen, auch Widgets genannt. Die Zielgruppe umfasst Personen mit keiner bis wenig Erfahrung in der Entwicklung von Web-Anwendungen.

Für diesen Zweck sind einige Grundlagen nötig. Als erstes wird der Begriff „End User Development“, die für die einfache Programmierung steht, und deren Konzepte für die Realisierung vorgestellt. Danach folgt die Erläuterung der programmierbaren Bausteine „Mashups“. Abschließend werden für die Realisierung die verwendeten Technologien beschrieben.

2.1 End User Development

Das Forschungsgebiet „End-User Development“ (EUD) (Lieberman u. a., 2006) beschreibt den Bereich der Softwareentwicklung für Personen (End-User), deren Hauptaufgabe nicht die Software-Entwicklung ist, die jedoch Domänenwissen aus dem Anwendungsfeld besitzen (EUSES). Diese „End-User“ oder auch Endbenutzer genannte Gruppe von Personen, steht im Fokus des Forschungsgebiets mit dem Ziel sie zu motivieren und ihnen die Programmierung zu vereinfachen.

Die Programmierung ist ein aufwendiger Prozess, welcher teilweise von Programmiersprachen unnötig verkompliziert wird, weil diese ohne sorgfältige Berücksichtigung der Problematik von Mensch-Computer-Interaktionen (HCI) entwickelt wurden (Chotirat u. a., 2000). Programmierer sind gezwungen Algorithmen und Daten auf Arten zu betrachten, die sich in anderen Kontexten völlig unterscheiden können. Der Begriff „natürlich“ bedeutet die genaue Repräsentation der Natur oder des Lebens, welche so funktioniert, wie der Menschen es erwartet bzw. sich vorstellt. Bei „Natürlicher Programmierung“ versteht man , Programmiersprachen und Entwicklungsumgebungen natürlicher bzw. näher an die Betrachtungsweise von nicht Programmierern zu gestalten (Myers u. a., 2004).

In „Natural programming languages and environments“ (Myers u. a., 2004) wird die Art der Programmierung folgendermaßen definiert: Der Prozess der Transformation eines mentalen Plans aus bekannten Begriffen in eine für den Computer kompatiblen Form. Je näher sich die

Programmiersprache an dem mentalen Plan des Entwicklers befindet, desto leichter ist der Transformationsprozess.

Das „Natural Programming Project“ (NatProg) verfolgt dabei die Strategie der Gestaltung von natürlichen bzw. intuitiven Werkzeugen, die die Komplexität und den Aufwand in der Programmierung reduziert. Diese Reduktion kommt meistens auf Kosten der Flexibilität in den Gestaltungsmöglichkeiten bei der Entwicklung zu Stande. Solch hoch spezialisierte Produkte ermöglichen jedoch selbst Anwendern, die nie zuvor ein Softwareprogramm geschrieben haben, mit einem verhältnismäßig geringen Aufwand spezielle Programme zu entwickeln. Bestehende Werkzeuge lassen sich auch durch die Optimierung von Entwicklungsprozessen intuitiver gestalten. Als Beispiel wird der Aspekt des Debuggings von Programmen durch die Neuerung „Whyline“ (Ko und Myers, 2004, 2008) vorgestellt. Bei falschem Programmverhalten fragt man sich häufig warum dies oder warum jenes nicht geschieht. Typische Entwicklungsumgebungen erlauben diese Art von Fragestellungen nicht, sondern stellen nur Variableninhalte und die Aufrufhierachie dar. Die „Whyline“ stellt nachvollziehbare Verbindungen zwischen Ereignissen und Programmcode her. Bei Fragen nach der Ursache eines Ereignisses, wird der relevante Programmcode und Programmfluss dargestellt, der dann vom Entwickler genauer untersucht werden kann. Alice ist ein Lehrwerkzeug zur Einführung erster Programmierkonzepte, das eine

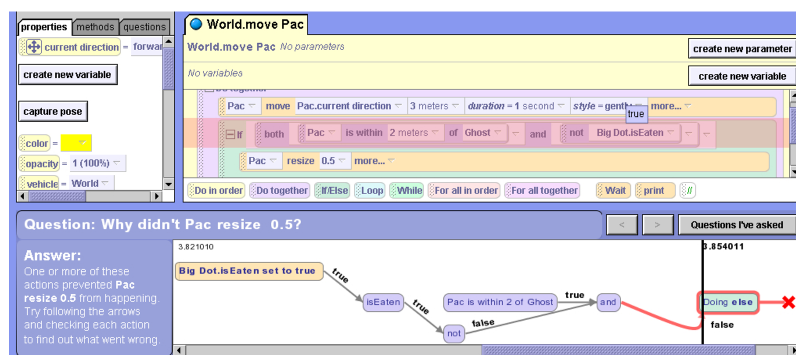

Abbildung 2.1: Alice WhyLine

Entwicklungsumgebung zur Erstellung von dreidimensionalen medialen Inhalten bereitstellt. Es verwendet erstmals das „Whyline“ Konzept, das den Programmfluss mit angereicherten Meta-Informationen grafisch dargestellt. Der Graph wird anhand von generierten Fragen[5] über Ursachen von Ereignissen und Eigenschaftsänderungen bestimmt. Die zur Auswahl stehenden Fragen werden durch statische und Laufzeit Analysen ermittelt. Der Graph stellt die Ereigniskette, die zu der gestellten Frage führt, und den relevanten Programmcode dar, die daraufhin weiter untersucht werden können (siehe Abb. 2.1). Untersuchungen mit Alice(Ko und Myers, 2004) zeigten das Fragen in Form von „Warum nicht?“ häufiger auftraten als „Warum ?“ und die Nutzung dieser Fragen eine schnellere Lokalisierung der Programmfehler erlaubt.

[5] Fragen in Form von „Warum ist die Farbe Gelb ?“ oder „Warum wurde diese Funktion ausgeführt?“

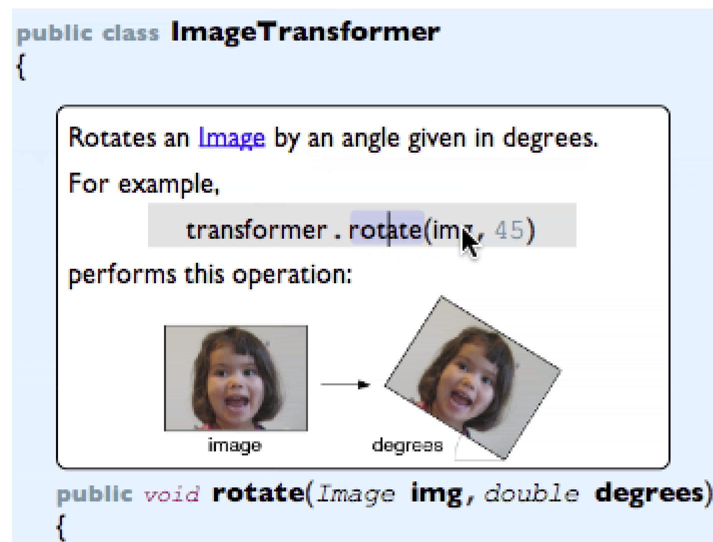

Abbildung 2.2: Barista: Eine mit Medien angereicherte Annotation einer Java-Methode

Ein anderes Beispiel zur Verbesserung des Entwicklungsprozesses, ist die Wahrnehmung und das Schreiben von Code in Editoren. Klassische Quelltext Editoren besitzen nur rudimentäre Funktionen wie Syntax-Hervorhebung und Kontext-Hilfe neben den Standardfunktionen der reinen Text-Editoren. Interaktive und visuelle Code-Editoren erweitern die klassischen Editoren, um eine strukturierte visuelle Repräsentation. Mit Eclipse[6] wurde der Standard weiter angehoben mit Funktionen wie Code-Vervollständigung, Code-Folding[7] und weiteren. Barista (Ko und Myers, 2006) ist ein Framework zur Erstellung von Editoren, die erlaubt, im Gegensatz zu anderen Frameworks und Editoren, erweiterte interaktive grafische Funktionen zu nutzen (siehe Abb. 2.2). Dies beinhaltet beispielsweise das verkleinern von Code-Fragmenten, alternative Darstellungen von Code-Fragmenten (z.B. Mathematische Formeln, boolesche Ausdrücke), In-Code Dokumentation via HTML, etc.

2.1.1 Konzepte

Für die Realisierung des Ziels des End-User Developments haben sich drei Konzepte gebildet:

- Visuelle Programmierung
- Programming by Example auch bekannt als Programming by Demonstration
- Domänenspezifische Sprachen

[6]Eclipse ist eine erweiterbare offene Entwicklungsplattform.
[7]Zusammenklappen von Code-Fragmenten

Jedes dieser Konzepte verfolgt einen anderen Schwerpunkt. Die visuelle Programmierung setzt den Fokus auf die visuelle Darstellung und Interaktion der Entwicklungsumgebung und Programmiersprache. „Programming by Example“ steht für die Programmierung durch die Demonstration einer Aufgabenbewältigung in einer Anwendung. Die Demonstration verwendet ausschließlich Bedienelemente der Anwendung, aus der später das resultierende Programm besteht und wiederholt ausgeführt werden kann. Domänenspezifische Sprachen stehen für anwendungsorientierte Programmiersprachen. Die Sprache wird dabei so konzipiert, dass sie kurz und prägnant Aufgaben in der konzipierten Anwendungsdomäne lösen kann.

Visuelle Programmierung

Bei der visuellen Programmierung (VP) kann der Benutzer durch das Zusammensetzen von vorgegebenen grafischen „Programmier-Bausteinen“ sein eigenes Programm programmieren. Die Bausteine bestehen soweit möglich aus Darstellungsstrukturen (Metaphern) aus der Anwendungsdomäne, wodurch die Verständlichkeit und die intuitive Bedienung verbessert wird. Die Strategien, die bei der VP verfolgt werden, sind:

- Konkretheit - Konkrete Objekte anstatt abstrakter Klassen
- Direkte Manipulation
- Explizitheit - Vermeidung impliziter Annahmen
- Sofortiges Feedback

Im Gegensatz zur textuellen Programmierung besitzt der erstellte Code in der VP potentiell eine höhere Aussagekraft, aufgrund der vielfältigen Eigenschaften die der mehrdimensionale Charakter ermöglicht. Diese Vielfältigkeit ist aber auch die Ursache der Problematik in der formalen Handhabung. Die textuelle Programmierung nutzt lexikalische Grundeinheiten aus einfachen Symbolen (Zeichen oder Worten), wo die syntaktische Beziehung zwischen den Symbolen ausschließlich durch die Hintereinanderaufschreibung festgelegt wird. Bei der VP hingegeben können die geometrischen Relationen (z.B. Position, Größe, ...), topologische Relationen (z.B. Vernetzung, Verschachtelung, ...) und dynamische Aspekte (z.B. Bewegung, Blinken, ...) eine wichtige Rolle spielen, die teilweise schwer mit Formalismen zu erfassen sind.

Green und Petre (1996) analysieren die Nutzbarkeit von „Visual Programming“ Entwicklungsumgebungen und stellt dabei fest, das bei der Entwicklung und Nutzung der Werkzeuge immer Kompromisse eingegangen werden müssen. Stefan Schiffer bewertet die visuelle Programmierung anhand von fünf Thesen und bezeichnet den Nutzen wie folgt (Schiffer, 1996, Seite 21):

> Die Stärken der VP liegen in speziellen Anwendungsgebieten, wo überschaubare und abgegrenzte Problemstellungen durch visuelle Metapher gut erfassbar sind. . . . Ebenfalls von hohem Wert sind graphische Darstellungen softwaretechnischer Sachverhalte in Form von Entwurfsskizzen und Visualisierungen, wenn auf Details zugunsten der Verständlichkeit verzichtet wird.

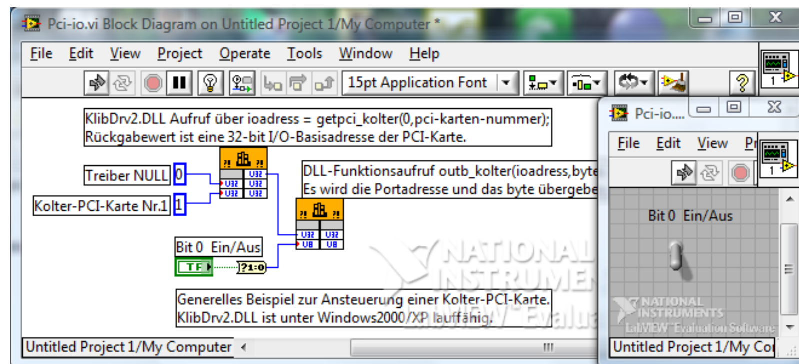

Abbildung 2.3: LabVIEW Datenfluss und Bedienelemente

LabVIEW und Croquet sind beispielhafte Umsetzungen der Programmier-Metapher der visuellen Programmierung die hier kurz vorgestellt werden. Es existieren weitere interessante Anwendungen wie Marten[8] oder PureData[9], auf die jedoch nicht näher eingegangen wird.

LabVIEW wird hauptsächlich von Wissenschaftlern und Ingenieuren zur Erstellung von Mess-, Prüf-, Steuer- und Regelsystemen verwendet. Für die Programmierung werden grafische Symbole mit hohen Wiedererkennungswert aus der Anwendungsdomäne eingesetzt und miteinander verbunden, so dass ein Datenfluss modelliert wird. Das Datenmodell beinhaltet Funktionsblöcke mit Ein- bzw. Ausgängen, die miteinander verbunden werden können. Bei der Ausführung des Programms, erhält man je nach verwendeten Funktionsblock ein Bedienelement für die Steuerung und Auswertung (siehe Abbildung 2.3).

Das Croquet SDK eine 3D Multibenutzer-Entwicklungsumgebung und verwendet eine von Smalltalk abgeleitete Programmiersprache namens Squeak. Mit ihr lassen sich virtuelle gemeinschaftliche Echtzeit-Welten entwickeln und verteilen. Programmiert wird textuell wie in altbekannten Entwicklungsumgeben, jedoch kann man in einer dreidimensionalen Welt mit sofortigem Feedback (siehe Abbildung 2.4) gemäß „WYSIWYG - What You See Is What You Get“ (Was du siehst, ist was du bekommst) programmieren.

[8] MacOSX Version einer visuellen Software-Entwicklungsumgebung: Marten
[9] PureData’s grafische Programmierung der Verarbeitung von Audio- und Video-Daten

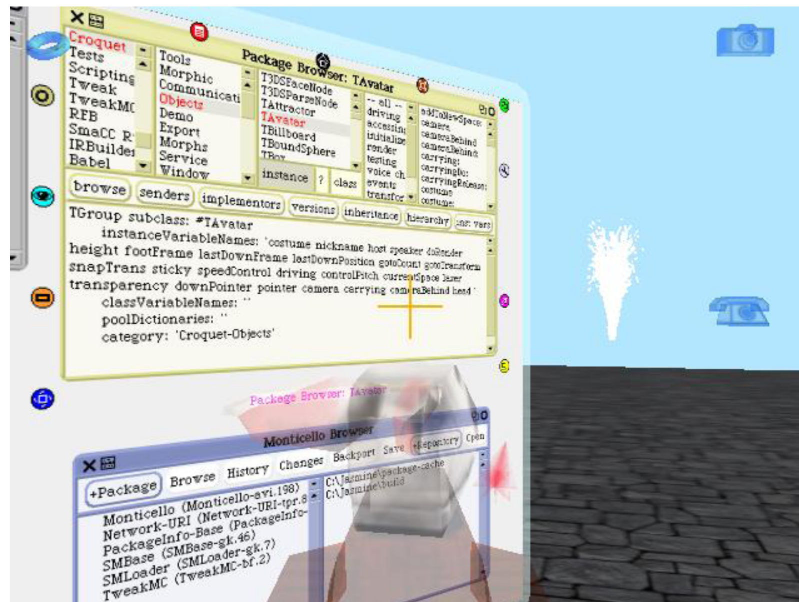

Abbildung 2.4: Croquet Quelltext Editierung

Programming by Example

Der Begriff „Programming by Demonstration“ (PBD) bzw. „Programming by Example“ (PBE) (Cypher u. a., 1993; Lieberman, 2001) wurde erstmals von Daniel Conrad Halbert (Halbert, 1984) beschrieben. Er besagt, dass

- Wenn der Benutzer ein Programm in einem „Programming by Example“ System schreibt, dann sind die Aussagen in seinem Programm dieselben, wie die Befehle, die er normalerweise dem System geben würde. Somit programmiert er in der Benutzerschnittstelle des Systems.
- Ein Programm wird geschrieben durch das Merken der Sequenz der Aktionen des Benutzers, während er normale Befehle erteilt. Dadurch programmiert der Benutzer mittels eines Beispiels, was das Programm machen soll.

Die Erscheinungsform der Benutzer-Programmierung reicht von Systemen mit einer Auswahl von Optionen bis hin zu Systemen die eine Programmierung als integraler Bestandteil erfordern:

- Das System bietet zur Individualisierung eine Auswahl von Optionen aus denen der Benutzer wählen kann. Zum Beispiel könnte ein Programm, welches druckt, eine Option für einfachen oder doppelten Zeilenabstand anbieten.

- Andere Systeme lassen den Benutzer Programme erstellen, welche aus Tastenanschläge oder anderen einfachen Benutzeraktionen bestehen. Die daraus resultierenden, ausführbaren Sequenzen werden Makro genant (Bsp.: Makro-Rekorder von UltraEdit[10]). Der Vorteil dieser Programmiertechnik ist, dass der Benutzer in der Benutzerschnittstelle programmiert, welche ihm bereits geläufig ist. Somit entfällt das Erlernen einer neuen Programmiersprache.
- Einige Systeme bieten eine separate Programmiersprache für die Automatisierung von Operationen an. Sei es die Schriftform der gesprochenen Sprache oder eine Obermenge der Benutzerschnittstellen-Kommandos, das wesentliche Merkmal ist die nähe zur bereits vertauten Benutzerschnittstelle.
- Abschließend gibt es Systeme, die eine Programmierung des Benutzers in der Benutzerschnittstelle als integraler Bestandteil besitzen. Tabellenkalkulationen sind gute Vertreter dieser Klasse. Sie erfüllen keine Aufgabe, solange der Benutzer kein Programm angibt, welche aus Formeln und Werten in Zellen besteht.

Damit eines der Systeme als PBE gilt, müssen sie noch in der Lage sein Aktionen aufzuzeichnen und abzuspielen.

PBE ist kein universell einsetzbares Konzept für die Programmierung. Cypher und andere (Cypher u. a., 1993; Halbert, 1984) grenzen im wesentlich den Nutzen von „Programming by Example“ auf wiederholbare Aktionen ein:

> Probably the largest potential use for programming by demonstration is for automating repetitive activities.

Domänenspezifische Sprachen

Domänenspezifische Sprachen (DSL) charakterisiert Paul Hudak in „Modular Domain Specific Languages and Tools“ (Hudak, 1998) als eine Programmiersprache die maßgeschneidert für eine besondere Anwendungsdomäne ist. Charakteristika von einer optimalen DSL ist die Fähigkeit, ein vollständiges Anwendungsprogramm für eine Domäne schnell und effizient zu entwickeln. Eine DSL ist nicht (notwendigerweise) universell einsetzbar. Im Gegenteil, es sollte präzise die Semantik einer Anwendungsdomäne einfangen und nicht mehr und nicht weniger. Beispiele für DSL sind SQL als Abfragesprache für relationale Datenbanken, Mathematica für mathematische Berechnungen, Csound für Klangkompositionen und Yacc und Lex als syntaktische und lexikalische Analysewerkzeuge für den Compilerbau. Ein Satz des Psychologen Abraham Maslow beschreibt den Sinn der domänenspezifischen Sprachen zutreffend als:

> If the only tool you have is a hammer, you tend to see every problem as a nail.

[10] http://www.ultraedit.com/

Wenn einem nur der Hammer als Werkzeug zu Verfügung steht, so wird jedes Problem als Nagel betrachtet und drauf geschlagen. Selbst beim Zusammensetzen der filigranen Einzelteile einer Uhr, wobei dann die Uhr nicht mehr die Uhrzeit korrekt anzeigen wird.

Skriptsprachen (Ousterhout, 1997) fallen unter die Kategorie DSL, sind aber als solche schwierig zu definieren, da sie sich mit der Zeit weiterentwickeln. Angefangen mit JCL[11] bis hin zu Perl, die in ihrer Mächtigkeit den *Third Generation Languages* (3GL) oder auch *System Programmiersprachen* genannt immer näherkommen. Wesentliche Merkmale einer Skriptsprache sind:

- Skriptsprachen sind Programme die bei der Ausführung von einem Skript-Interpreter interpretiert werden.
- Die Syntax und Semantik ist fehlertolerant ausgelegt. Eine eingeschränkte Anzahl von Schlüsselwörtern soll die Verwendung für den Benutzer vereinfachen.
- Aufgrund von Vereinfachungen ist die Nutzung oftmals auf ein Anwendungsgebiet eingeschränkt.

Das Ziel von Skriptsprachen ist, im Gegensatz zu den 3GL, im wesentlichen nur die Verknüpfung von bestehenden Komponenten. Sie werden auch manchmal als *Glue Language* (Klebstoff-Sprache) oder *System Integration Language* (System Integrationssprache) bezeichnet.

Domänenspezifische Sprachen eignen sich als Konzept für die Erstellung einer Widget Orchestrierung für End-User sehr gut. Die kompakte Syntax und Semantik reduziert den Aufwand bei der Erlernung der Sprache und ermöglicht mit kurzen und prägnanten Ausdrücken das gewünschte Verhalten zu bezeichnen. Weiterhin lassen sich mit der Verwendung mächtiger Werkzeuge die Formulierung von Ausdrücken erleichtern.

2.2 Mashup

Mashup ist ein Begriff aus der Musikbranche, welcher sich mit dem Web 2.0 Hype auch in der Informatik etabliert hat. Es bezeichnet das Mischen von mehreren Musikstücken von verschiedenen Interpreten zu einem neuen Musikstück. Allgemein ausgedrückt werden Inhalte aus unterschiedlichen Ursprüngen zu einem neuen Inhalt kombiniert. In der Informatik wird darunter die Kombination von Web-Inhalten bezeichnet. Die Technik erlaubt, abhängig vom eingesetzten Produkt, unerfahrenen Endbenutzern eigenständig anspruchsvolle Webseiten zu gestalten. Im

[11] Job Control Language ist die Steuersprache für Stapelverarbeitungen in einem Großrechnerumfeld. Aufgabe der JCL ist es, die auszuführenden Programme, deren Reihenfolge, sowie eine Laufzeitumgebung (Verbindung zu physischer Hardware, E/A und Dateien) vorzugeben (Quelle Wikipedia).

Gegensatz zu Produkten zur Webseiten-Erstellung, die Laien nur erlauben Inhalte optisch anspruchsvoll darzustellen, erlauben Mashup-Produkte dem Laien neben der optisch anspruchsvollen Darstellung auch die Realisierung komplexer Logik. Die Anwendungsentwicklung durch Mashups ist ein vielversprechender Ansatz im Bereich des *End-User Developments*.

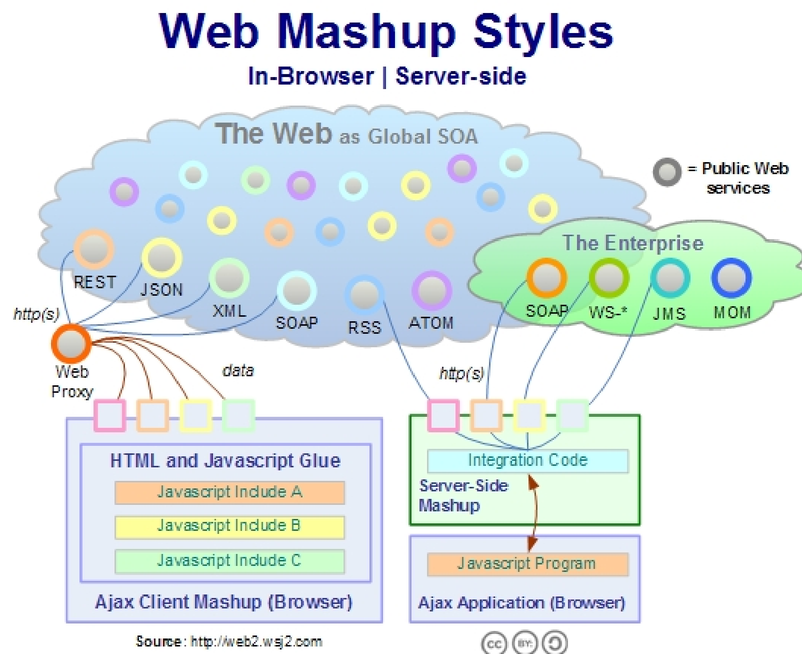

Abbildung 2.5: Mashup Struktur

Mashups sind Web-Anwendungen, die ihre Daten maßgeblich aus verschiedenen Quelle beziehen. Der Zugriff auf die Daten, ist über eine Vielzahl von Schnittstellen möglich (REST, Webservices, Datenbanken, . . .). Die zusammengetragenen Daten werden so verarbeitet, dass daraus aufgewertete, weiterverwendbare Daten produziert werden. Die Daten werden entweder über eine grafische Schnittstelle präsentiert (Widget), die auch Interaktionen erlaubt, oder sie werden in einem einfachen Datenformat (XML, RSS[12], ...) exportiert und die Präsentation wird dem Webbrowser überlassen. Die Mashup-Logik kann entweder auf dem Server oder auf dem Client (Webbrowser) laufen. Die Webbrowser basierten Mashups nutzen Technologien wie HTML, AJAX, Flash usw. für die Implementierung der Widgets (siehe Abb. 2.5). Ein simples Widget zur Währungsumrechnung ist in Abbildung 2.6 zu sehen.

Die Definition von Mashup ist nicht sehr präzise, wodurch schon die bloße Verknüpfung des Sozial-Netzwerkes „StudiVZ“[13] mit dem Fotoalbum „Flickr“[14] als Mashup gilt.

[12] RSS ist ein Format für die einfache und strukturierte Veröffentlichung von Änderungen auf Web-Seiten
[13] `www.studivz.de`
[14] `www.flickr.com`

Mashups, deren Widgets eine hohe Funktionsvielfalt und eine reichhaltige Anwendungslogik beeinhalten, gehören zur Familie der *Rich Internet Applications* (RIA). Sie sind den klassischen Desktop-Anwendungen sehr ähnlich.

Abbildung 2.6: Beispiel Widget Interaktion: Währungsrechner

Die Anwendungsentwicklung durch Mashups ist ein vielversprechender Ansatz im Bereich des *End-User Developments*. Das Internet als zugrunde liegende Plattform erlaubt den Zugang zu stetig wachsenden und öffentlich zugänglichen Diensten. Durch die Nutzung dieser Dienste, verschiebt sich der Entwicklungsaufwand von der traditionellen Programmierung hin zum Auffinden von High-Level Diensten, das Verbinden dieser Dienste und das Erstellen von Benutzerschnittstellen. Diese Aktivitäten erfordern weniger Details als die Programmierung, sind potentiell näher an der Anwendungsdomäne und kann besser durch Werkzeuge, aufgrund stärkerer Restriktionen, unterstützt werden.

Es existieren viele Mashup-Entwicklerwerkzeuge die unterschiedliche Zielgruppen, von dem interessierten Gelegenheits-Surfer bis hin zu dem erfahrenen Web-Entwickler, anvisieren. Sie erlauben es End-User ohne bzw. mit geringen Programmierkenntnissen im Privaten alltagstaugliche und in Firmen kurzfristige, situationsbedingte Web-Anwendungen zu erstellen. So kann entweder über eine Software-Entwicklungsumgebung (z.B. Eclipse) mit Hilfe von Programmierkenntnissen oder in einem Mashup-Editor benutzerfreundlich und ohne Programmierkenntnisse ein Mashup erstellt werden.

Die Mashup-Entwicklerwerkzeuge erlauben eine schnelle Erstellung von Mashups und da Mashups im Grunde genommen Web-Anwendungen sind, können sie dem Paradigma „Rapid Web Application Development“ zugeordnet werden, welche im Kern für eine schnelle Entwicklung mit geringen Kosten von Web-Anwendung steht.

Die Entwicklung eines Mashups erfordert die Implementierung der drei Komponenten des Mashups (siehe Abb. 2.7):

- Daten-Lieferant
- Daten-Transformation / Mashup Ausführung
- Daten-Präsentation / Benutzerschnittstelle (Widget)

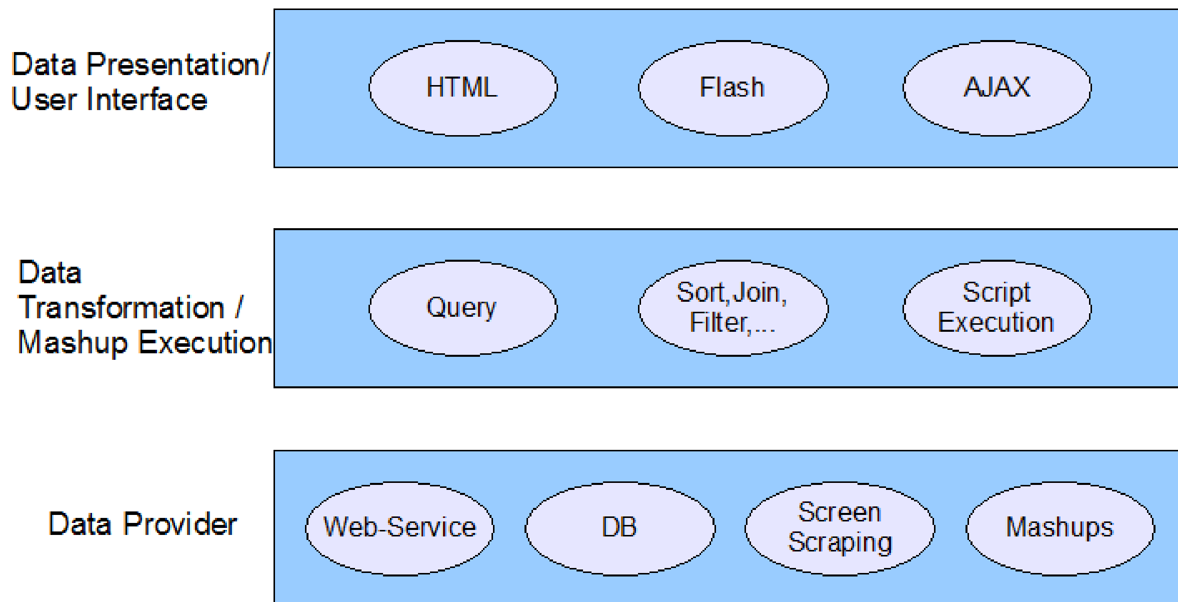

Abbildung 2.7: Aufbau eines Mashups

Bevor in der Komponente *Daten-Transformation* Ressourcen verarbeiten werden können, müssen diese erstmal über die Komponente *Daten-Lieferant* zur Verfügung gestellt werden. Der Aufwand der Bereitstellung der Daten ist abhängig von der Quelle, aus der sie bezogen wird. So sind Webservices weit verbreitet und werden von vielen Werkzeugen gut unterstützt und sind dadurch einfach zu integrieren, wogegen Daten in komplexen HTML-Seiten einen aufwendigen, individuellen und fehlertoleranten Extraktionsalgorithmus[15] benötigen. Neuartige Datenquellen, die im Werkzeug keine Unterstützung besitzen, können oftmals mit Programmierkenntnissen nachträglich ergänzt werden.

Die Daten-Transformation wird entweder textuell mittels einer Programmiersprache oder einem grafischen Editor realisiert. Die textuelle Umsetzung ist meist flexibler und setzt eine erhöhte Einarbeitungszeit voraus. Die Implementierung durch einen grafischen Editor reduziert den Aufwand im Vergleich zur textuellen Programmierung, aber auf Kosten der Funktionsfähigkeit.

Die Implementierung der Daten-Präsentation kann entfallen, sofern auf fertige Templates zurückgegriffen oder die Darstellung dem Webbrowser überlassen werden kann. Komplexe Wid-

[15] Zu extraktierende Daten können z.B. bei Zeichenketten über reguläre Ausdrücke oder bei XML-Interpretation über XPath[16] ausgedrückt werden und müssen bei Strukturänderungen des Dokuments trotzdem zuverlässig arbeiten.

[16] Die XML Path Language (XPath) ist eine Abfragesprache und erlaubt Teile eines XML-Dokumentes zu adressieren.

gets erfordern Fachwissen in der Web-Entwicklung (HTML, Javascript[17], ...) und bleibt dem unerfahrenen End-User verwehrt. Einfache Widgets (Formulare, minimale Interaktionsmöglichkeiten) können vom End-User über grafische GUI-Designer ohne besondere Fachkenntnisse erstellt werden.

Eine vollständige Mashup-Entwicklungsplattform muss dem Entwickler bei der Implementierung der drei Komponenten unterstützen. Die Auswahl an Produkten ist jedoch sehr groß und auf unterschiedliche Zielgruppen fixiert, weshalb ein Teil nur ein oder zwei Komponenten unterstützt.

2.2.1 Widget

Widget ist ein sehr allgemeiner Begriff und umfasst die Bezeichnung von Steuerelementen in grafischen Benutzeroberflächen und Mini-Anwendung für Webbrowser und Desktop. Charakteristisches Merkmal ist die Eigenschaft eine grafische Komponente in einem größeren System zu sein, die auf Benutzereingaben reagieren kann. In dieser Arbeit wird der Begriff Widget auf die Webbrowser eingegrenzt. Ein Widget ist eine kleine Web-Anwendung die typischerweise in Webseiten eingebettet ist. Widgets besitzen meistens eine geringe Funktionsvielfalt, wodurch sie oftmals nicht als professionelle sondern als nützliche Anwendungen im Sinne eines Werkzeugs wahrgenommen werden. Es existieren unzählige Widget Spezifikationen (Opera, Yahoo, Stardock, ...) jedoch werden hier nur drei genannt, die eine Standardisierung zum Ziel haben:

- OpenSocial Gadget ehemals Google Gadget XML (OpenSocial Gadget)
- OpenAjax Metadata 1.0 Specification Widget Metadata (OpenAjax Widget)
- W3C Widget Packaging and Configuration (W3C Widget)

OpenSocial Gadgets (OS-G) ist eine Entwicklung von Google und erlaubt die vollständige Implementierung eines Widgets in HTML und Javascript oder in Flash. Layout- und Verwaltungs-Informationen werden deklarativ in XML angegeben. OS-G erlaubt die Benutzersteuerung eines Widgets über vordefinierte „UserPrefs“ die automatisch von einem OS-G Interpreter in HTML-Eingabeelemente dargestellt werden. Es ist nicht möglich, über die XML-Beschreibung eine Gadget-to-Gadget Schnittstelle zu definieren.

OpenAjax Metadata 1.0 Specification Widget Metadata (OAM-SWM) verwendet XML um Meta-Eigenschaften eines Widgets deklarativ zu definieren und erlaubt die vollständige Implementierung eines Widgets in HTML und Javascript. Die Spezifikation wurde offiziell im Mai 2010 verabschiedet und aufgrund der Überlappung des Schreibens dieser Arbeit, wird sie hier nur

[17] JavaScript ist eine Skriptsprache, die hauptsächlich für das DOM-Scripting in Webbrowsern eingesetzt wird (Wikipedia).

am Rande aufgeführt und Informationen wurden nachträglich am Ende der Fertigstellung der Arbeit hinzugefügt. OAM-SWM bietet zwei Techniken, die eine Inter-Widget Kommunikation erlaubt. Variablen werden in OAM-SWM über „Property“-Elemente definiert, die mit dem Attribut „sharedAs“ und einem global eindeutigen Bezeichner öffentlich deklariert werden. Alle Widgets, die den gleichen globalen Bezeichner verwenden, arbeiten mit der selben globalen Variable. Der zweite Ansatz verfolgt das Publish/Subscribe Muster[18] und wird über das Element „topic“ gesteuert. Die Attribute „publish“ und „subscribe“ besagen ob Werte gelesen oder geschrieben werden sollen. Die Logik des Topics, wann Daten gesendet und wie empfangene Daten ausgewertet werden, wird über Javascript für jedes einzeln implementiert.

Die W3C Spezifikation setzt ihren Fokus auf Widgets, die außerhalb eines Webbrowsers ausgeführt werden. Es beschreibt wie die Struktur aller benötigten Ressourcen (Bilder,Bibliotheken,HTML,...) in einer komprimierten ZIP-Datei gestaltet sein muss und die Widget Eigenschaften (Name,verwendete Ressourcen) in einer XML-Datei. Eine Unterstützung zur Inter-Widget-Kommunikation bietet die Spezifikation nicht.

Alle Widget Spezifikationen erlauben Standalone Widgets zu definieren, wobei das von W3C keine Nutzung von externen Ressourcen erlaubt. Sie müssen als Bundle mit dem Widget ausgeliefert werden. Die Fähigkeit zur Inter-Widget-Kommunikation wird nur von der vor kurzem veröffentlichten Spezifikation „OpenAjax Metadata 1.0 Specification Widget Metadata“ unterstützt, die aufgrund der späten Erscheinung nicht in die Analyse mit einbezogen wurde.

2.2.2 Mashlet

Der Begriff Mashlet wird aufgrund ihrer Wortbedeutung in der Literatur wie auch im Internet unterschiedlich definiert. Das *Mash* stammt aus dem englischen und bedeutet mischen. Die Endung *let* kann von verschiedenen Begriffen abgeleitet werden (z.B. Servlet für Server und Applet) und eine entsprechende Bedeutung besitzen.

In „OMOS: A Framework for Secure Communication in Mashup Applications“ (Zarandioon und et al., 2008) wird der Begriff Mashlet als ein sichtbares HTML-Konstrukt bestehend aus einem Javascript-Service bezeichnet, welches von einer Domäne kontrolliert wird und seine Daten bezieht. Konzeptuell werden Mashlets mit Prozessen und Daemons[19] im Betriebssystem und mit Webservices gleichgesetzt. JackBe, ein Hersteller eines Mashup-Entwicklerwerkzeuges, definiert den Begriff als eine benutzerorientierte Micro-Anwendung, die die Visualisation bzw. das Gesicht eines Mashup in einer Webseite darstellt. Im wesentlichen wird das Mashlet als die

[18] Publish / Subscribe (Publizieren / Abonnieren) beschreibt den Nachrichtenaustausch mehrerer Beteiligter. Der Publisher sendet seine Nachrichten nicht gezielt an Subscriber sondern an eine Gruppe in der Subscriber Teilnehmer sind.

[19] Computerprogramm das im Hintergrund arbeitet

visuelle Präsentation bzw. grafische Benutzerschnittstelle einer Mashup-Anwendung definiert, welches auch unter den Begriff Widget und Gadget bekannt ist.

In dieser Arbeit bezeichnet der Begriff Mashlet eine Menge von Widgets, die in irgendeiner Form miteinander Daten austauschen und aufeinander reagieren können.

Das Prinzip der Mashup-Entwicklung kann von der Datenebene kann auch auf der Präsentationsebene (Widget) angewandt werden. Die Nutzung und Kombination von fertigen Widgets und Mashlets, erlaubt es dem End-User mit einfachen Mitteln neue informations- und interaktionsreiche Mashlets zu erstellen. Die Produkte Intel Mash Maker und IBM Mashup Center erlauben die Erstellung von Mashlets, jedoch ist die Ausführung auf das Produkt beschränkt und eine offene Beschreibung der Mashlets zur Wiederverwendung in externen Anwendungen gibt es nicht. Desweiteren lassen sich die erstellten Mashlets nicht rekursiv als Bausteine in weiteren Mashlet Entwicklungen einsetzen.

Ein Mashlet besteht aus den folgenden Komponenten:

- Widget Repository - Liste der verwendeten Widgets
- Wiring - Publish / Subscription der Widgets
- Export - Öffentliche Schnittstelle
- Layout - Anordnung der Widgets

Das *Widget Repository* benennt die Widgets die als Bausteine zur Verfügung stehen und die miteinander verknüpft werden können. Das *Wiring* definiert die Verknüpfung der Widgets und realisiert dadurch die Mashlet-Logik. Der *Export* bezeichnet die Funktionen des Mashlets, die von außen z.B. einem anderen Mashlet genutzt werden können. Dies erlaubt eine rekursive Wiederverwendung. Das *Layout* bestimmt die Anordnung und die Größe der eingesetzten Widgets und kann durch ein HTML Dokument vorgegeben werden.

2.2.3 Orchestrierung

Der Begriff Orchestrierung (engl. Orchestration) erscheint im Zusammenhang eines Orchesters. Ein Orchester besteht aus vielen Musikern, die von einem Dirigenten geleitet werden (siehe Abb. 2.8). Der Dirigent ist für die technische Koordination der Musiker verantwortlich (z.B. Taktvorgabe). Die Musiker kennen nur ihren Part und achten nur auf den Dirigenten.

In der klassischen Informatik wird der Begriff in dem Paradigma „Serviceorientierte Architektur“ (SOA) (Endrei u. a., 2004) verwendet und beschreibt einen ausführbaren Geschäftsprozess

[20]Quelle: Wikipedia

Abbildung 2.8: Orchester: Mescheder Windband, 2008 [20]

aus der Sicht eines Teilnehmers. Er definiert die Geschäftslogik und die Reihenfolge der ausführbaren Aktionen und bestimmt die Interaktionen zwischen den verschiedenen Services bzw. Web Services auf der Nachrichtenebene (Peltz, 2003). Ein Service ist sich nicht bewusst, dass er Teil einer Kollaboration ist. Änderungen an der internen Implementierung eines Services bleiben der Öffentlichkeit verborgen. Die öffentliche Schnittstelle bleibt unberührt. Ein wichtiger Aspekt in der Orchestrierung ist die Kontrolle des Prozessablaufs über einen zentralen Koordinator.

Im Kern bedeutet die Orchestrierung die Koordination mehrerer unabhängiger Teilnehmer durch einen Leiter.

Mashups nutzen das Konzept der Orchestrierung. Die Funktion des Leiters wird von der Laufzeitumgebung des Mashups übernommen und die Teilnehmer bzw. Services sind andere Mashups oder Web Services, die als Datenquellen herangezogen werden. Die Geschäftslogik entspricht der Daten-Konsumierung und -aufbereitung.

Das Konzept spiegelt sich auch in der Mashlet Entwicklung wieder. Die Web Services korrespondieren mit den Widgets. Widgets interagieren miteinander über den Austausch von Daten (Nachrichten) durch einen Koordinator und wissen nicht, das sie Teil einer Orchestration sind. Mashlets beschreiben jedoch keinen Geschäftsprozess, sondern definieren eine aus Widgets bestehende Anwendung.

2.2.4 Produkte

Die Mashup-Entwicklerwerkzeuge sind teilweise sehr unterschiedlich hinsichtlich ihrer Funktionen und Bedienungsfreundlichkeit. Einige konzentrieren sich auf einzelne Komponenten des

Mashups und andere wiederum berücksichtigen alle. Die Anzahl der Produkte ist zu groß, um sie hier alle aufzuführen, weshalb nur ausgewählte Vertreter vorgestellt werden.

Intel Mash Maker (beta)

Das im Beta-Stadium befindliche Produkt Intel Mash Maker, ist eine Webbrowser-Plugin basierte Lösung zur Erstellung und Nutzung von Mashups mit dem Ziel das Surfverhalten zu verbessern. So lassen sich für einzelne Web-Seiten zusätzliche Informationen über Widgets oder Mashups einbetten, die Intel Mash Maker als Mashup bezeichnet. Die an Web-Seiten gebundenen Mashups werden automatisch beim Besuch der jeweiligen Seite geladen und aktiviert. Dieser transparente Vorgang verbessert das Surferlebnis.

Unterstützung der Mashup-Komponenten:

- Daten-Lieferant - Rudimentär: Über ein Screen-Scraping[21] Verfahren wird das Template strukturiert und dabei sichtbare Elemente in Gruppen eingeteilt und URL-Parameter typisiert.
- Daten-Transformation - Rudimentär: Bei der Einbettung der Mashups lassen sich die zu darstellenden Daten über ein Menü steuern. So kann z.B. bei einer Suchmaschine bestimmt werden, ob die ganze Seite dargestellt werden soll oder ob nur die Suchergebnisse aufgelistet werden sollen. Komplexere Transformationen lassen sich darüber nicht umsetzen.
- Daten-Prästentation - Rudimentär: Neue Widgets lassen sich direkt in Javascript und der Widget API[22] von Intel Mash Maker programmieren oder durch das Werkzeug unterstützend als Templates von bestehenden Web-Seiten erstellen.

Intel Mash Maker erlaubt Werkzeug gestützt die Erstellung jeder Komponente, jedoch ist die Umsetzung auf nur jeweils ein Verfahren beschränkt. Datenquellen wie Datenbanken, XML oder Webservices lassen sich nicht nutzen. Bei der Datentransformation wird nur die Filter-Funktion unterstützt und die Daten-Präsentation erlaubt nur die Erstellung von Templates von Web-Seiten. Die verwendete Technologie erlaubt zwar eine Erstellung von komplexen Mashups, jedoch müssen diese aufwendig in klassischer Form in Javascript und HTML programmiert werden. Das Werkzeug leistet dabei keine Hilfestellung.

Intel Mash Maker unterstützt den Entwickler bei der Entwicklung von Mashlets durch das EUD Konzept „visuelle Programmierung“. Mashups bzw. Widgets werden visuell platziert und verschoben und die Konfiguration der Widgets (Wiring) findet über den Einsatz von Assistenten statt. Unterstützung der Mashlet-Komponenten:

[21] Manuelles untersuchen und extrahieren von Informationen einer Internet-Seite zur Bildung eines automatischen Verfahrens

[22] Application Programming Interface kurz API

- Widget Repository: Integrierte Suche des Repository des Herstellers. Erweiterbar durch die Community.
- Wiring: Assistenten gesteuerte Verdrahtung durch Drop-Down Listen.
- Export: Nicht vorhanden.
- Layout: Visuelle Platzierung mittels Drag&Drop.

WidgetBox

WidgetBox hat sich auf die Erstellung von Widgets spezialisiert. Widgets können entweder über vordefinierte parametrisierbare Templates (Youtube, Flickr, Twitter, Vimeo, ...) erstellt werden, oder durch die Nutzung von Javascript und HTML oder Flash programmiert werden. Die Erstellung eines Widgets über Templates geschieht über Assistenten die eine Anzahl von Parametern abfragt, darunter welche Farben, Zeichensatz, Größe, Benutzerdaten für Youtube etc. Der Assistent erlaubt auch die Nutzung weiterer Templates, die in Tabs untergebracht sind. Wird ein Widget über Javascript oder Flash erstellt, so entfällt die Unterstützung durch Assistenten und alles muss in dieser Sprache umgesetzt werden.

Unterstützung der Mashup-Komponenten:

- Daten-Lieferant: Abhängig vom Template werden über Assistenten Quellen eingebunden.
- Daten-Transformation: Nicht vorhanden.
- Daten-Prästentation - Rudimentär: Neue Widgets lassen nur anhand vordefinierter Templates erstellen und minimal anpassen.

WidgetBox erlaubt nicht die Erstellung von Mashlets.

IBM Mashup Center 2.0

Das IBM Mashup Center ist eine Enterprise-Mashup-Plattform bestehend aus den drei Komponenten InfoSphere MashupHub, Lotus Mashups und Catalog. Das InfoSphere MashupHub dient dazu die unterschiedlichsten Ressourcen aus dem Web oder Enterprise (Datenbanken, Excel, CSV, XML, SAP, ...) abzurufen, zu verarbeiten und für die Nutzung im IBM Mashup Center im REST-Stil zur Verfügung zustellen. Die Funktionen werden im Rapid Application Development Stil über Assistenten und grafische Editoren bereitgestellt.

Die Komponente Lotus Mashups ist die Plattform, auf der die Widgets verwendet werden. Es stellt für den End-User eine Arbeitsfläche bereit, auf der sich Widgets platzieren und miteinander in Beziehung setzen lassen. Einzelne Datenelemente von Widgets können gezielt an andere Widgets verschickt werden (Wiring). Die Konfiguration der Interaktion geschieht über einen Assistenten, der jeweils typisierte öffentliche Daten des zu sendenden und empfangenden Widget zur Auswahl anbietet (siehe Abb. 2.9). So kann zum Beispiel das Widget Kundenliste verwendet werden, um die Adresse eines ausgewählten Kunden auf dem Widget Straßenkarte darstellen zu lassen. Neue Widgets werden durch Assistenten oder extern in HTML, Javascript und OpenAjax Metadata 1.0 Specification Widget Metadata (OpenAjax Widget) erstellt.

Das Catalog stellt das Repository für die Verwaltung aller Daten, Widgets und Mashups, die im IBM Mashup Center genutzt werden können. Weiterhin erlaubt es die Datenquellen zu filtern, aggregieren und zu sortieren.

Unterstützung der Mashup-Komponenten:

- Daten-Lieferant: Unterstützt eine Vielzahl von Schnittstellen über das InfoSphere MashupHub.
- Daten-Transformation: Grafischer Editor über Catalog.
- Daten-Prästentation - Rudimentär: Neue Widgets lassen nur in Form von Tabellen oder von externen Web-Seiten erstellen über Assistenten erstellen.

Das IBM Mashup Center nutzt für die Erstellung von Mashlets das EUD Konzept der visuellen Programmierung. Es stellt eine leere Arbeitsfläche bereit, die der Benutzer mit über das Catalog ausgewählten Widgets befüllen kann. Die Platzierung ist dem Benutzer überlassen und lässt sich nachträglich mittels Drag&Drop ändern. Welche Daten zwischen welchen Mashups ausgetauscht wird, wird vom Anwender über einen Assistenten bestimmt. Dabei werden einige Auswahllisten mit validen Optionen angeboten, aus dem der Benutzer wählen kann. Unterstützung der Mashlet-Komponenten:

- Widget Repository: Integrierte Suche des Repository. Erweiterbar durch den Anwender.
- Wiring: Assistenten gesteuerte Verdrahtung durch Drop-Down Listen.
- Export: Nicht vorhanden.
- Layout: Visuelle Platzierung mittels Drag&Drop.

JackBe Presto 2.7

JackBe Presto 2.7 ist eine Enterprise-Mashup-Plattform, die die Erstellung, Verwaltung und Ausführung von Mashups erlaubt. Datenquellen können über eine Vielzahl von unterstützten Schnittstellen eingebunden werden. Die Mashup-Logik, sprich die Verarbeitung der Daten, kann entweder über den visuellen Editor Presto Wires (Abb. 2.10) oder textuell in der offenen XML basierten Sprache (EMML) implementiert werden. Die Funktionsweise des visuellen Editors Presto Wires ist vergleichbar mit dem von IBM Mashup Center. Methoden und Datenquellen stehen als Komponenten bereit, die als visuelle Bausteine auf einer Arbeitsfläche platziert werden können. Visuell erstellte Linien beschreiben den Datenfluss zwischen den Bausteinen und die Datenverarbeitung innerhalb eines Bausteins wird über Assistenten konfiguriert. Mit der Bereitstellung des Editors Presto Wires unterstützt JackBe Presto das EUD Konzept der visuellen Programmierung.

Die Sprache EMML ist eine DSL, die speziell zur Beschreibung des Prozessflusses eines Mashups konzipiert wurde und die Portabilität von Mashup Design und die Interoperabilität von Mashup Lösungen erlaubt. Dies wird durch die Verfügbarkeit eines reichhaltigen High-level Mashup Domänen Vokabular zur Konsumierung und Transformation von verschiedenen Web Datenquellen (REST, WSDL[23], RSS/ATOM, RDBMS[24], ...). Weitere unterstützte Funktionen des Vokabulars betreffen das Filtern, Sortieren, Joinen, Gruppieren, Aggregieren usw. von Daten, das Einbetten von Script (Javascript, JRuby, ...), bedingte Anweisungen und parallele Verarbeitung. Sie beschränken sich jedoch auf die Datenverarbeitung und sagen nichts über die visuelle Präsentation der Daten und der Interaktionsmöglichkeiten aus. Für eine leichtere Entwicklung in EMML gibt es das Eclipse Plugin Presto Mashup Studio. JackBe unterstützt damit die beiden EUD Konzepte „Visuelle Programmierung“ und „domänenspezifische Sprache“ und spricht darüber eine breite Zielgruppe an.

Unterstützung der Mashup-Komponenten:

- Daten-Lieferant: Unterstützt eine Vielzahl von Schnittstellen.
- Daten-Transformation: Grafischer Editor Wires oder über die textuelle Erstellung durch EMML in der Software-Entwicklungsumgebung Eclipse.
- Daten-Prästentation - Rudimentär: Neue Widgets lassen sich nur anhand vordefinierter Templates erstellen und minimal über Parameter anpassen.

JackBe Presto 2.7 erlaubt nicht die Erstellung von Mashlets.

[23] Die Web Services Description Language (WSDL) ist eine plattform-, programmiersprachen- und protokollunabhängige Beschreibungssprache für Netzwerkdienste (Webservices) zum Austausch von Nachrichten auf Basis von XML (Wikipedia).

[24] Relationales Datenbankmanagementsystem kurz RDBMS

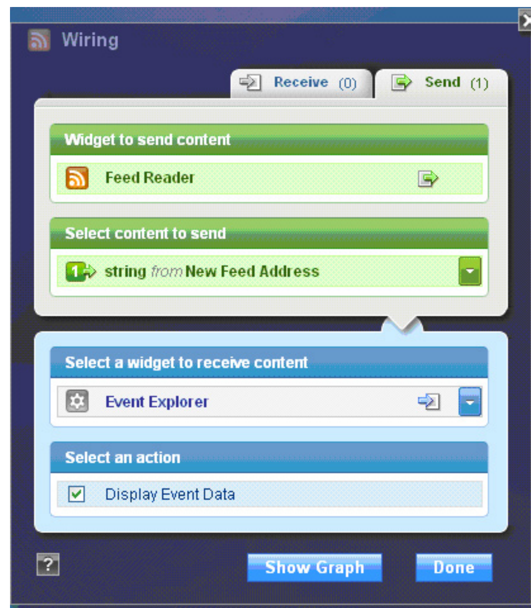

Abbildung 2.9: IBM Mashup Center: Wiring

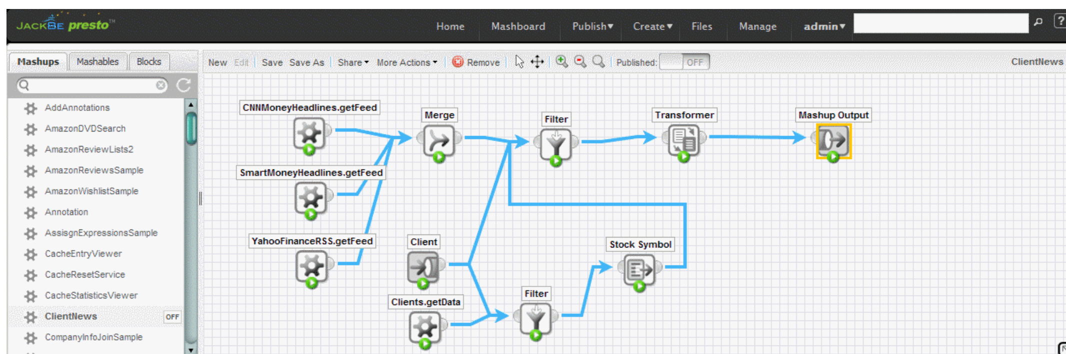

Abbildung 2.10: JackBe Presto Wires (Quelle: JackBe)

Google

Anfangs bot Google den Google Mashup Editor (GME) an, um über Javascript, HTML, CSS[25] und XML Feedinhalte abzufragen und zu manipulieren. Dieses Werkzeug basiert hauptsächlich auf eine textuelle Programmierung. Im späteren Verlauf stellte Google GME ein und begann die Entwicklung an der Google App Engine, welcher Teil der Google Cloud Computing ist, in die alle bisherigen Erkenntnisse aus GME einflossen.

Die Entwicklung von Mashups für die Google App Engine geschieht über Python, Java oder einem JVM[26]-basierten Interpreter oder Compiler wie Ruby, Javascript usw. Die Implementierung erfolgt somit traditionell textuell über eine IDE und richtet sich an erfahrene Software-Entwickler. Dadurch bietet sich somit keine Neuerungen für End-User, außer speziell angepasste Bibliotheken für Web-Anwendungen. Google wird hier nur erwähnt, weil es als Global Player in vielen Bereichen des Internets aktiv ist.

Google bietet für die Widget bzw. Mashlet Entwicklung die Spezifikation namens Google Gadget an. Widget Eigenschaften werden deklarativ in XML ausgedrückt und die Logik und das Layout werden entweder referenziert oder direkt als Code eingebettet. Google Gadget unterstützt HTML/Javascript, Silverlight und Flash.

Google App Engine richtet sich aufgrund der traditionellen Softwareentwicklung an professionelle bzw. erfahrene Softwareentwickler und ist somit für End-User Development ungeeignet.

Zusammenfassung

Viele Produkte nutzen bei der Mashup-Entwicklung ein eigenes Widget Format und erlauben teilweise einige Widgets aus populären Anwendungen (z.B. das von Google Gadget) zu importieren. Andere wie JackBe Presto 2.7 ignorieren oder unterstützen nur rudimentär die Widget-Erstellung und befassen sich ausschließlich mit der Datenerfassung und -aufbereitung, die als wiederverwendbare Services bereitgestellt werden. Die Nutzung von mehreren Widgets und die Gestaltung der Interaktion zwischen ihnen werden nur von wenigen Produkten wie Intel Mash Maker und IBM Mashup Center unterstützt. Das Format jedoch ist proprietär und somit für Außenstehende nicht zugänglich.

Die vielen Widget Formate sind so ausgelegt, das sie die Anforderung der jeweiligen Anwendung erfüllen. Da die wenigsten eine Widget-Interaktion vorsehen, enthalten die Widget-Formate auch keinerlei Informationen über eine mögliche öffentliche Schnittstelle zur Interaktion. Die Produkte, die eine Widget Interaktion bzw. Orchestrierung erlauben, nutzen ein

[25] Cascading Style Sheet (CSS) ist eine deklarative Sprache für strukturierte Dokumente die festlegt, wie ein besonders ausgezeichneter Inhalt oder Bereich dargestellt werden soll.

[26] Java Virtual Machine

proprietäres Format. Das ursprüngliche Format von IBM Mashup Center liegt mittlerweile in der Obhut der Organisation OpenAjax und wurde im Mai 2010 veröffentlicht, der für die Verwendung in dieser Arbeit zu spät erschien. Es fehlt somit ein öffentliches Widget-Format, das für eine Widget Orchestrierung genutzt werden kann. Weiterhin ist das Format zur Widget Orchestrierung ebenfalls proprietär. Somit sind Widget Orchestrierungen an eine Plattform gebunden. Desweiteren lassen sie sich nicht als wiederverwendbare Komponenten in weiteren Widget Orchestrierungen nutzen.

3 Analyse

Ziel dieser Arbeit ist es ein System zu entwickeln, welches End-User ohne Erfahrungen in irgendeiner Programmiersprache erlaubt bestehende Widgets zu einem Mashlet zu verknüpfen, um neue Funktionen bereitzustellen wie sie beim Intel Mash Maker und IBM Mashup Center möglich sind. Zusätzlich sollen diese Mashlets als wiederverwendbare Komponenten in Form von Widgets zur Verfügung stehen. Im folgenden wird ein Anwendungsszenario vorgestellt, die das Ziel der Arbeit konkretisiert. Desweiteren werden die Anforderungen an das System definiert und die technischen Randbedingungen bei Realisierung erörtert. Weiterhin werden bestehende Widget Standards in Hinblick auf den Einsatz in einer Mashlet-Anwendung untersucht. Der Einsatz von EMML steht nicht zur Auswahl, da es sich ausschließlich mit der Daten-Transformation und -Bereitstellung befasst. Sie sagt nichts über die Daten-Präsentation aus.

3.1 Anwendungsszenario

3.1.1 Organizer - Zeitplaner

Die Aufgabe eines Organizers ist die Strukturierung und Verwaltung von Daten. Im alltäglichen Gebrauch wird er oftmals für die Verwaltung von Adressen, Telefonnummern und Terminen ver-

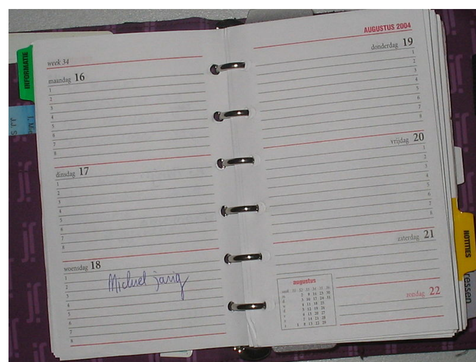

Abbildung 3.1: Zeitplaner Quelle: M. Minderhoud

wendet. Die Erscheinungsformen sind vielfältig (Ringordner, PDA, Computer), jedoch besitzen alle die gleichen Merkmale:

- Eingabemöglichkeit - Stift, Tastatur
- Suchfunktion - Blättern, explizite Suche nach Kriterien
- Daten-Aufbewahrung - Papier, Speicher

Der Zeitplaner (siehe 3.1), auch Terminplaner genannt, befasst sich ausschließlich mit Terminen. Für die Aufgabenbewältigung sind folgende Funktionen erforderlich:

- Erstellung eines Termins
- Löschung eines Termins
- Bearbeitung eines Termins
- Suche eines Termins nach Datum

Für die Aufgabenbewältigung ist es notwendig, passende Widgets zu finden, die die Funktionen übernehmen können. Widgets besitzen unterschiedlich viele Funktionen und können abhängig von ihrer Funktionsvielfalt mehrere spezialisierte Widgets ersetzen. Es existiert nicht für jede gewünschte Funktion ein passendes Widget und muss entweder über die Kooperation mehrerer Widgets abgebildet werden oder gezielt in einer beliebigen Mashlet kompatiblen Technologie implementiert werden. Die Entscheidung über die Umsetzung eines Mashlet obliegt alleine dem Entwickler, dem End-User.

Die gewünschten konkreten Funktionen der Anwendung „Organizer" könnten folgendermaßen lauten:

- Erstellung eines Termins - Der Anwender klickt auf einen leeren Eintrag, wodurch das Datum an der Position und ein Standard-Text den Termin bilden.
- Löschung eines Termins - Ein Doppelklick auf einen Termin löscht diesen.
- Bearbeitung eines Termins - Bearbeitet wird immer der aktuell ausgewählte Termin. Die Selektion findet über einen Klick auf einen existierenden Termins statt.
- Suche eines Termins nach Datum - Das Blättern der zeitlich sortierten Termine.

Die Abbildung 3.2 symbolisiert einen Organizer und zeigt wie das Zusammenspiel der einzelnen Mashlet-Komponenten miteinander aussehen könnte. Auffällig ist das unstimmige Look&Feel der Mashlets. Jedes dieser Mashlets hat seine eigene Funktion, die auch im Kontext eines normalen Organizers auftauchen. Der Kalender wird benötigt, um ein genaues Datum auszuwählen. Das Mashlet „Timeline" kann in diesem Beispiel als optional betrachtet werden,

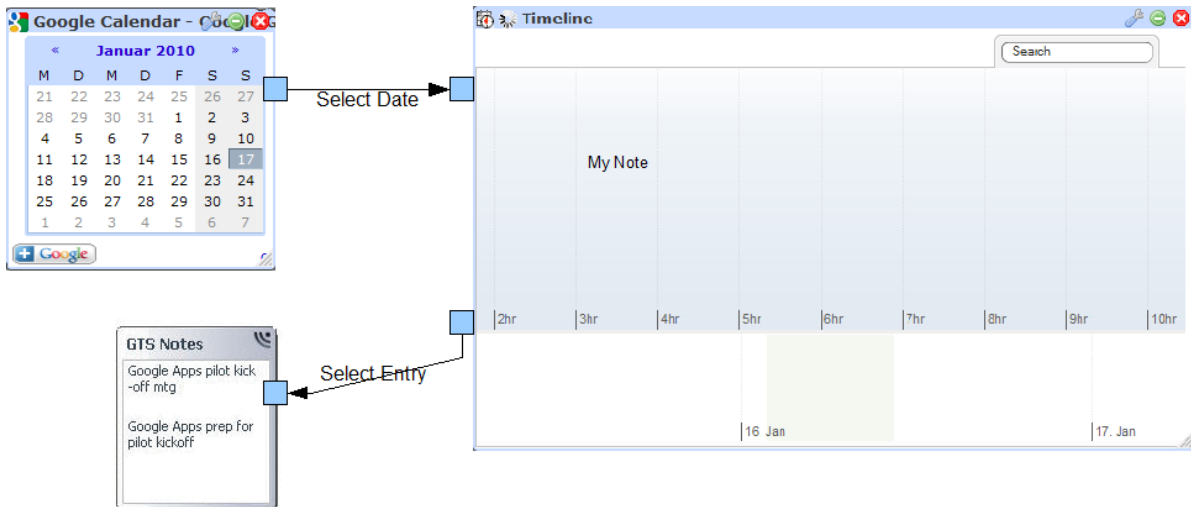

Abbildung 3.2: Trivial Organizer aus einzelnen Mashlet-Komponenten

jedoch erlaubt es uns eine weitere Unterteilung des Tages in Stunden. Das Mashlet „GTS Notes“ dient als Lese- und Schreibutensil der Nachrichten. Die Verknüpfung und der Austausch von Informationen untereinander, ergeben einen Organizer.

Dieses Organizer Mashlet lässt noch erweitern. Der Ort des Termins ließe sich auf einer Karte darstellen und gegebenfalls eine Route automatisch berechnen lassen, sofern die aktuelle Position ermittelbar ist.

3.2 Anforderungen

3.2.1 Nichtfunktionale Anforderungen

Sutcliffe (2005); Sutcliffe u. a. (2003) beschreibt wesentliche Anforderungen an ein Werkzeug für den End-User aus der Kosten-Nutzen Perspektive. End-User sind beschäftigte Menschen, die Werkzeuge nutzen, um ihr Ziel effizienter zu erreichen.

1. Geringer Lernumfang - Ein neues Werkzeug erfordert einen Initialaufwand für die Einarbeitung, die vom End-User als Hürde wahrgenommen wird.
2. Wiederverwertbarkeit - Ist das Werkzeug nur für seltene Probleme einsetzbar, so stellt sich die Frage nach der „globalen“ Effizienz.

3. Fehleranfälligkeit - Die Folgen von Fehler bei der Anwendung oder beim Lernen beeinflussen den End-User stark. Eine Folge ist die Demotivierung und Zweifel über die Sinnhaftigkeit des Werkzeugs.
4. Sicherheit - Sicherheitsrelevante Daten vor Unbefugten schützen
5. Skalierbarkeit&Performance bei Wiederverwendung und Parallelisierung
6. Mandantenfähigkeit
7. Portabilität

3.2.2 Funktionale Anforderungen

Das im Kapitel 3.1 vorgestellte Szenario weist mehrere Eigenschaften auf, die Funktionalität bezogen sind. Dieses Kapitel fasst diese zusammen und stellt somit die funktionalen Anforderungen an das Mashlet-System dar.

Die Hauptaufgabe des Mashlet-Systems ist die kontrollierte Interaktion zwischen verschiedenen Widgets (Interoperabilität), die auf verschiedenen Technologien aufbauen (Integration). Die Form der Interaktion findet über einen Datenaustausch statt. Wer welche Daten verschickt und empfängt, wird vom Entwickler, dem End-User, bestimmt (Kontrollierte Kommunikation). Nicht jedes Widget ist vertrauenswürdig und darf Zugang zu allen Daten besitzen. Im besonderen darf der Widget-Code benachbarte Widgets oder das Mashlet-System verändern (Sicherheit). Erstellte Anwendungen müssen pausierbar sein und zu einem beliebigen Zeitpunkt fortgesetzt werden können (Persistente Daten). Desweiteren können solche Anwendungen von mehreren Personen gleichzeitig genutzt werden (Mandantenfähigkeit) und durch weitere Widgets oder Mashlets erweitert werden, ohne die interne Implementierung ändern zu müssen (Rekursive Wiederverwertbarkeit und Erweiterbarkeit).

1. Integration und Nutzung bestehender Widgets
2. Widget Orchestrierung
    - Interoperabilität - Die Möglichkeit Widgets miteinander zu verknüpfen, um neue Aufgaben zu lösen
    - Kontrollierte Kommunikation
3. Persistente Daten
4. Rekursive Wiederverwertbarkeit und Erweiterbarkeit von Widget Orchestrierungen

Diese funktionalen Anforderungen sollen nicht als vollständige Beschreibung des Funktionsumfangs verstanden werden. Sie haben lediglich einen exemplarischen Charakter.

3.3 Technische Randbedingungen

Für die Realisierung der Anforderungen und im besonderen die der funktionalen, müssen die technischen Randbedingungen erörtert werden. Viele Widgets wurden als Closed Box Anwendungen[27] realisiert, was zum Problem der Wiederverwendung bezüglich der Verknüpfung zu einem Mashlets führt. Denn solche Anwendungen bieten entweder keine öffentliche Schnittstellen an oder verwenden gezielt Sicherheitskonzepte um einen Zugriff auf interne Daten zu verhindern. Gehen wir vom Organizer Beispiel in Abbildung 3.2 aus. Ursprünglich weist das Kalender Objekt nur die Funktion der Darstellung des aktuellen Tages auf. Im Kontext des Organizers wird die Auswahl eines beliebigen Tages benötigt. Vorzugsweise mit einer farblichen Hervorhebung der Auswahl. Diese Information muss exportierbar sein, d.h. dass der ausgewählte Tag abgerufen und an einer anderen Stelle weiterverwendet werden können muss. Um bestehende Widgets verwenden zu können, müssen diese an eine standardisierte Schnittstelle angepasst werden. Eine Vorgehensweise wird in Abschnitt 3.3.1 erläutert.

Das Widget „Timeline“ besitzt eine offene Schnittstelle, die das Manipulieren von Einträgen erlaubt, jedoch bietet sie keine Möglichkeit an, um zu erfahren, für welchen Eintrag sich der Anwender interessiert. Manipulationen an Einträgen müssen persistent gehalten werden, denn bei einem Neustart müssen die aktuellen Termine weiterhin verfügbar sein. Das Widget selbst ist dazu nicht in der Lage. Die Persistierung muss daher durch eine externe Entität realisiert werden.

Die Nutzung bestehender Widgets führt somit zu folgenden Problemen:

- Fehlende bzw. unzureichende Schnittstelle, um extern Daten auszulesen oder zu schreiben
- Nicht standardisierter Zugang
- Persistente Daten

3.3.1 Wiederverwendbare Widgets

Viele bereits verfügbare Widgets bieten, wenn überhaupt, nur einen eingeschränkten Zugang zur internen Verarbeitung (Events, Daten, ...) erlauben. Existiert eine Schnittstelle, so ist sie abhängig von der verwendeten Technologie des Widgets. Dieser Zugang ist jedoch äußerst wichtig für die Verknüpfung von Widgets. Deshalb muss die mögliche Isolation aufgebrochen werden und dazu bieten sich im wesentlichen zwei Ansätze an:

- Data Injection

[27] Abgeschlossene in sich isolierte Anwendungen

  - GET Parameter, HTML Input Elemente
  - Via Document Object Model (DOM)
- Code Injection
  - Hooks (HTML Events z.B. onselect)
  - Via DOM

Hauptaufgabe des „Data Injection“ ist das Setzen eines bestimmten Zustands. Zum Beispiel lassen sich die Werte von HTML Input Elementen[28] ändern, indem das jeweilige DOM Element verändert wird. Die Aufgabe des „Code Injection“ ist die Schaffung einer Abfragemöglichkeit des internen Zustands. Die Schnittstelle wird so erweitert, das benötige Informationen abgefragt werden können oder es werden dafür gezielt Event-Handler installiert.

Da viele Widgets aus HTML-Konstrukten mit Javascript-Snippets bestehen, sind die zwei genannten Ansätzen realistisch anwendbar. Jeder moderne Webbrowser unterstützt die Abbildung des HTML-Dokuments als DOM, wodurch eine Manipulation über diese Schnittstelle sich anbietet. Die Verwendung der DOM-Schnittstelle führt jedoch zwei Probleme ein:

- Webbrowser Inkompatibilitäten: Historisch bedingt unterscheiden sich einige Umsetzungen von Schnittstellen wie der von DOM, wodurch bei der Entwicklung einer Web-Seite auf Inkompatibilität Rücksicht genommen werden muss.
- Webbrowser Sicherheit: Nicht jeder Teil des Dokuments ist für jeden zugänglich. Dieser Aspekt wird im Abschnitt 3.3.1 genauer erläutert.

Zu Anfangszeiten des Internets, herrschte ein reger Wettbewerb, um den meist benutzten Webbrowser. Jeder Hersteller fügte Funktionen ein, die der der Konkurrenz übertrumpfen sollten. Dabei wurden bewusst offene Schnittstelle abgewandelt implementiert, um den Webbrowser inkompatibel zu Anderen zu machen und so die Anwender zwingen für ein bestimmtes Produkt zu entscheiden. Dieser Konkurrenzkampf ist heute noch wahrnehmbar und spiegelt sich z.B. in der API der DOM-Schnittstelle wieder. Die Nutzung der Abfragesprache XPath ist zwischen dem Internet Explorer und Mozilla Firefox unterschiedlich implementiert. Gängige Praxis in der Handhabung dieser Problematik, ist die Bildung einer einheitlichen Schnittstelle als Fassade, die je nach Webbrowser die entsprechende Implementierung nutzt. Alternativ kann auf eine Vielzahl von Bibliotheken zugegriffen werden, die diese Problematik berücksichtigen.

[28] Solche Elemente umfassen Eingabefelder für Text, Auswahlisten, ...

Webbrowser Sicherheit - Cross Site Scripting

Bei der Entwicklung von Widgets wurde nicht explizit die Möglichkeit vorgesehen, die eine Manipulation am Funktionsumfang (Neue Funktionen, Events) oder Inhalt erlaubt. Einige wurden sogar so entwickelt, dass eine Manipulation verhindert werden soll. Eine Technik verwendet dazu das „IFrame“ HTML-Element, um den Webbrowser dazu zu veranlassen, den Inhalt isoliert von anderen Elementen zu verarbeiten. Das IFrame benötigt eine URL eines HTML-Dokuments, das geladen und an dieser Position dargestellt wird. Findet ein Zugriff über z.B. Javascript vom Haupt-HTML-Dokument statt und unterscheiden sich die Domänen der URL's voneinander, so versteckt der Webbrowser den Inhalt des IFrame.

Widgets stammen typischerweise von verschiedenen Herstellern und somit von unterschiedlichen Domänen, wodurch das Isolationsmodell als Sicherheitskonzept zu einem Problem wird. Bestehende Lösungsvorschläge setzen eine bewusste Unterstützung zur Kommunikation zwischen Widgets über einen Vermittler voraus oder der Manipulation der Domäne der HTML-Seiten. Das OpenAjax Hub (OpenAjax Hub) bietet eine offene Javascript Bibliothek, die über ein Publish/Subscribe Verfahren den Austausch von Daten zwischen Komponenten innerhalb einer HTML-Seite erlaubt. Sie erlaubt auch die Kommunikation mit Komponenten innerhalb eines IFrame's ohne das Sicherheitskonzept vollständig auszuhebeln. Der nachträgliche Einbau der Funktionalität der Kommunikation erfordert jedoch die Erweiterung der Original HTML-Seite, die das Isolationsmodell verhindert und ist somit nicht zu empfehlen.

Eine Alternative ist das Vorgaukeln der Ursprungsdomäne, wodurch das Isolationsmodell im Webbrowser nicht angewendet wird. Dazu müssten alle externen Aufrufe im Webbrowser abgefangen werden und über einen Proxy, der in der gleichen Domäne residiert, umgeleitet werden. Diese Lösung ermöglicht jedoch genau das, was das Sicherheitskonzept verhindern sollte: Der Zugang Dritter an sensible Daten. Dieser Lösungsvorschlag arbeitet auf Vertrauensbasis. Vertrauenswürdige Widgets und Manipulationen am Original können über Zertifikate oder ähnliche Mechanismen identifiziert werden.

Die Einführung eines Standard zur Inter-Widget Kommunikation bei der Widget Erstellung, würde das Sicherheitsproblem auf eine einfache und saubere Art entschärfen.

3.4 Mashlet

Die Mashlet Entwicklung erfordert eine einheitliche Widget Definition und Schnittstelle zur Interaktion sowie die Definition des Zusammenspiels der Widgets. Mit diesen Informationen lässt sich eine webbasierte plattformunabhängige Widget Orchestrierung realisieren. In diesem Kapitel werden die bestehenden Spezifikationen in Hinblick auf die notwendigen Anforderungen analysiert.

3.4.1 Widget Definition

Aufbauend auf den Erkenntnissen der Standard Widget Beschreibungen aus 2.2.1, den allgemeinen Anforderungen in 3.2 und den technischen Notwendigkeiten und Hürden aus 3.3.1, werden hier die Anforderungen an die Widget Definition als Baustein der Mashlet Erstellung genannt. Es müssen mindestens folgende Informationen in einer Widget Definition enthalten sein:

- Name - Name zur Identifikation.
- Ressourcen - Alle Ressourcen müssen eingebettet oder über Referenzen erreichbar sein.
- Code - Der Widget Code muss entweder eingebettet oder über Referenzen erreichbar sein.
- Layout - Das Widget Layout muss entweder eingebettet oder über Referenzen erreichbar sein.
- Öffentliche Schnittstelle - Typisierte Methoden oder Variablen um Anwendungsbezogene Daten zu ermitteln oder zu setzen. Sie erlaubt die externe Kommunikation mit dem Widget und ermöglicht die Interaktion mit anderen Widgets.
- Integrations Code - Spezieller Code für die Einbettung externer Widgets.

Der *Name* wird verwendet um ein Widget eindeutig referenzieren zu können. Eine Auflistung der *Ressourcen* erlaubt Teile des Widgets auszulagern und als wiederverwendbare Komponenten zu nutzen. Der *Code* steht für die Anwendungslogik, dem Herzen des Widgets. Dieser kann ausgelagert oder in der Definition mit eingebettet sein. Das *Layout* beschreibt das Aussehen, dass direkt in HTML oder einer anderen Sprache bestimmt wird. Die *öffentliche Schnittstelle* ist entscheidend für die Interaktionsfähigkeit und somit für die Erstellung eines Mashlets, denn sie gibt vor was mit dem Widget kommuniziert werden kann. Findet die Kommunikation über Variablen statt, so müssen zusammengesetzte Datentypen unterstützt werden, um Nachrichten mit hohen Informationsgehalt verschicken zu können. Als Beispiel besteht beim Organizer eine Dateneinheit aus einem Datum, einer Uhrzeit und einem Text. Sollen externe Widgets eingebunden werden, so müssen diese speziell über einen *Integrations Code* angepasst werden. Dieser berücksichtigt eventuelle besondere Initialisierungen, Daten-Normalisierungen und das Mapping der öffentlichen Schnittstelle auf die internen Widget Daten.

OpenAjax Metadata 1.0 Specification Widget Metadata

Die „OpenAjax Metadata 1.0 Specification Widget Metadata“ spezifiziert wie ein Widget definiert und ausgeführt wird. Sie wurde erst im Mai 2010 offiziell verabschiedet und die nachträgliche kurzfristige Analyse ergibt, dass sie allen Widget Anforderungen gerecht wird. Sie nutzt XML als Datenformat und alle Widget Eigenschaften werden über Elemente und Attribute beschrieben. Darunter fallen Widget Bezeichner, abhängige Ressourcen, Integrations Code und öffentliche Schnittstellen. Die Tabelle 3.1 listet Widget Anforderungen und wie die Spezifikation diese erfüllt.

Anforderung	Details
Name	Ja, über Attribut *Name*
Ressourcen	Ja, über Element *Library*
Code	Ja, über Element *Javascript*, *Library* oder *Require*
Layout	Ja, über Element *Content*
Öffentliche Schnittstellen	Variablen über *Property* oder *Topic*
Integrations Code	Ja, über *Javascript*, *Library* oder *Require*.

Tabelle 3.1: Erfüllung der Anforderungen der Widget Definition von OpenAjax Metadata 1.0 Specification Widget Metadata

W3C Widget Packaging and Configuration

Die „W3C Widget Packaging and Configuration“ beschreibt das Widget mitsamt aller Ressourcen und dessen struktureller Aufbau einer komprimierten ZIP[29] Datei. Eigenschaften wie Name, verwendete Ressourcen und Verarbeitungshinweise werden in einer XML hinterlegt. Es erfüllt alle Widget Anforderungen bis auf „Intergrations Code“. Die Tabelle 3.2 listet alle Anforderungen, inklusive einer kurzen Erläuterung, auf.

OpenSocial Gadget

Das „OpenSocial Gadget“ nutzt XML um ein Widget zu beschreiben. Es erlaubt über eingebettetes HTML und Javascript ein Widget zu laden und zu manipulieren, jedoch stellt es keine Vorrichtung um eine externe Schnittstelle für die Inter-Widget-Kommunikation zu definieren. Es lassen sich jedoch die definierten Benutzereingaben durch *UserPref* dazu zweckentfremden. Problematisch ist die geringe Auswahl an Datentypen und die fehlende Unterstützung

[29] ZIP ist ein Kompressionsalgorithmus

Anforderung	Details
Name	Ja, über Element *Name*
Ressourcen	Nur Referenzen über *Feature* oder *Icon*
Code	Nur Referenz über *Content*
Layout	Nur Referenz über *Content*
Öffentliche Schnittstellen	Variablen über *Preference*
Integrations Code	Nein

Tabelle 3.2: Erfüllung der Anforderungen der Widget Definition von W3C Widget

zusammengesetzter Datentypen. Ein weiteres Problem ist die potentielle Darstellung der Benutzereingaben über die Laufzeitumgegebung. Die Spezifikation sieht vor, dass alle *UserPref* über eine GUI verändert werden können. Dadurch würden erstellte Widgets für die Mashlet Entwicklung unerwartete Nebeneffekte in anderen Laufzeitumgebungen bilden. Das „OpenSocial Gadget“ erfüllt somit bedingt alle Widget Anforderungen, denn die Nutzung von „UserPref“ als öffentliche Schnittstelle stört sich an der Tatsache, dass sie auch als automatisch generierte GUI-Elemente verwendet werden.

Anforderung	Details
Name	Ja
Ressourcen	Nur Referenzen über *ModulePrefs*
Code	Nur Referenz über *Content*
Layout	Nur Referenz über *Content*
Öffentliche Schnittstellen	Variablen über *UserPref*, jedoch auch Standard GUI-Elemente. Wenige Datentypen, keine zusammengesetzte Datentypen
Integrations Code	Indirekt über eingebetteten Code in *Content*

Tabelle 3.3: Erfüllung der Anforderungen der Widget Definition von OpenSocial Gadget

3.4.2 Orchestrierung

Die Orchestrierung ist das reibungslose Zusammenspiel mehrerer unabhängiger Teilnehmer, welche von einer zentralen Figur koordiniert werden. Der Entwickler, der End-User, koordiniert das Zusammenspiel mehrerer Widgets um eine von ihm als sinnvoll erachtete Aufgabe zu lösen. Das Zusammenspiel bezeichnet die Interaktionen zwischen Widgets, die durch den Austausch von Informationen ausgeführt werden. Folgende Mashlet Anforderungen werden benötigt:

- Name - Name zur Identifikation.

- Ressourcen / Widgets - Auflistung der verwendeten Widgets über deren Namen. Der Widget Name muss einzigartig sein.
- Widget Initialisierung / Instanziierung - Initiale Setzung von Eigenschaften wie z.B. Farbe, Titel Bezeichner, ...
- Nachrichtenfluss definieren - Beschreibt den Datenaustausch zwischen Widgets über deren öffentliche Schnittstelle.
- Öffentliche Schnittstelle - Schnittstelle zur Wiederverwertung in einem anderem Mashlet.

Das Mashlet muss über einen Namen referenziert werden, im Besonderen bei der Wiederverwendung in einem weiteren Mashlet. Eine Auflistung von Widgets dient als Fundus von Bausteinen für die Implementierung. Die Widgets müssen instanziiert und gegebenenfalls mit Parametern initialisiert werden. Die Instanzen produzieren und konsumieren Nachrichten über die öffentliche Widget Schnittstelle. Der Nachrichtenfluss muss vom Entwickler steuerbar sein und dient ihm als Mittel zur Implementierung des Anwendungsverhalten. Der Mechanismus zur Steuerung muss die gezielte Verteilung der Nachrichten an ausgesuchte Widgets erlauben. Da ein Mashlet auch als wiederverwendbare Komponente fungieren soll, muss es eine öffentliche Schnittstelle geben, die kompatibel zu der öffentlichen Widget Schnittstelle ist und die es ihr erlaubt als ein Widget in einer anderen Mashlet-Anwendung zu agieren kann.

Eine Analyse von IBM Mashup Center und Intel Mash Maker bezüglich der Orchestrierung von Widgets ist nur begrenzt möglich. IBM stellt keine öffentliche Spezifikation bereit und die Untersuchung der Demo-Anwendung ist nicht ergiebig gewesen. Das IBM Mashup Center erlaubt eine Exportierung eines Mashlet in Form von Atom Feeds[30]. Diese Feeds verweisen auf unzählige weitere Atom Feeds, wodurch eine Rekonstruktion der Funktionsweise sich als zu aufwendig gestaltet.

Intel Mash Maker weist bei der Speicherung eines erstellten Mashlets hin, das es öffentlich auf einen Server hinterlegt wird. Intel hat auch keine Informationen über die Struktur veröffentlicht. Der Zugang und die Analyse der Mashlet Struktur wird damit verwehrt.

3.5 Zusammenfassung

Der Kern der Arbeit betrifft die Bereitstellung einer offenen Sprache zur Orchestrierung von Widgets und Wiederverwendung als Komponente in einer Orchestrierung. Dazu wurde ein Anwendungsszenario geschildert, die Eigenschaften und die Anforderungen der Sprache erläutert.

[30] Atom ist ein XML-Standard der den plattformunabhängigen Austausch von Informationen ermöglicht.

Die Widget Orchestrierung ist eine noch nicht weit verbreitete Fähigkeit unter den Mashup-Werkzeugen und wird derzeit nur von Intel Mash Maker und IBM Mashup Center genutzt. Die zu orchestrierende Widget Komposition wird über den Webbrowser erstellt und verwendet speziell entwickelte Widgets. Die erstellte Komposition ist nur auf der erstellten Anwendung ausführbar. Eine Export-Funktion mit einem öffentlich dokumentierten Format existiert nicht. Die Fähigkeit, eine orchestrierte Widget Komposition als ein Widget bzw. Mashup zu verwenden und es in anderen Widget Kompositionen zu verwenden ist nicht vorgesehen. Es fehlt eine portable, von der Laufzeitumgebung[31] unabhängige, Beschreibungssprache zur Orchestrierung einer Widget Komposition die wiederum in einer Widget Orchestrierung eingesetzt werden kann. Dadurch ließe sich eine Anwendung mit dem Werkzeug der Wahl erstellen und auf allen kompatiblen Laufzeitumgebungen ausführen.

Bevor jedoch die Widget Orchestrierung angegangen werden kann, muss erstmal eine taugliche Widget Spezifikation als Standard ermittelt werden. Viele Widget Spezifikationen entstanden als maßgeschneiderte Lösungen zusammen mit der Anwendung in der sie eingesetzt werden und erfüllen deshalb nur die Anforderungen, die diese Anwendung mit sich bringt. Einige wenige werden weiterentwickelt (z.B. Google Gadget XML), mit dem Anspruch als Widget Standard dienen zu können, die jedoch den Schwerpunkt alleine auf der grafischen Benutzerschnittstelle zwischen Anwendung und Benutzer legen. Die explizite Unterstützung zur Inter-Widget-Kommunikation ist erst mit dem im Mai 2010 erschienen OpenAjax Metadata 1.0 Specification Widget Metadata möglich, jedoch kommt sie für diese Arbeit zu spät. Es fehlt somit eine Widget Spezifikation, die das Aussehen, Verhalten und die Schnittstelle zur Inter-Widget-Kommunikation beschreiben lässt. Es muss somit eine Widget Spezifikation entwickelt werden, die die Anforderungen der bestehenden Mashup-Produkte erfüllt und eine Schnittstelle zur Inter-Widget-Kommunikation aufweist.

[31] Mit der Laufzeitumgebung ist die Orchestration Engine gemeint, die die Orchestrierung ausführt und nicht der Webbrowser, auf dem ein Teil der Orchestration Engine läuft.

4 Design / Konzept

In diesem Kapitel wird das Design und ein allgemeiner Entwurf der im Kapitel 3 beschriebenen Eigenschaften und Anforderungen einer Widget Orchestrierung vorgestellt. Es wurden bereits in Kapitel 2 verschiedene Systemlösungen und Technologien für Mashlet-Entwicklungen erläutert, die jedoch die Erstellung von wiederverwendbaren Mashlets in Form von Widgets im Erstellungsprozess nicht erlauben. Desweiteren werden die Widget Spezifikationen, der in Kapitel 3.4.1 gestellten Anforderungen nicht gerecht und eine öffentliche Spezifikation zur Widget Orchestrierung (Mashlet) ist nicht verfügbar. Dieses Kapitel beschreibt die beiden Spezifikationen und die Mashlet Laufzeitumgebung.

4.1 Architektur

Die Architektur der Widget Orchestrierung ähnelt der Service orientierten Architektur (SOA), welche ein Strukturmuster ist, das auf die Minimierung direkter und fester Abhängigkeitsbeziehungen zwischen den Diensten eines verteilten Softwaresystems abzielt. Dienstnutzer ermitteln und binden zur Ausführungszeit anhand Eigenschaftsanforderungen in einem Dienstverzeichnis Dienstanbieter (Siehe Abb. 4.1). Die Widget Orchestrierung vermeidet direkte Ab-

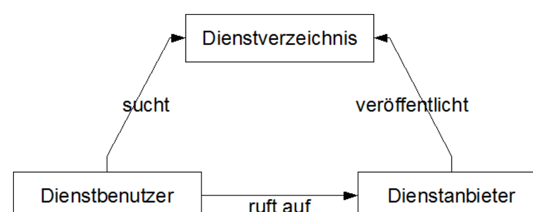

Abbildung 4.1: SOA Beziehungsstruktur: Dreieckmodell

hängigkeiten zwischen Widgets durch die Nutzung eines Blackboard-Modells[32]. Ein Widget

[32]Ein Blackboard-Modell basiert auf der Vorstellung einer Gruppe von Experten, die durch Zusammenarbeit ein Problem lösen, das sich aufgrund seiner Komplexität der Lösung durch einen einzelnen Experten entzieht. Auf dem Blackboard werden dabei von einzelnen Teilprozessen Daten in einer hierarchisch organisierten Form abgelegt. Das Blackboard ist nun in der Lage, andere Teilprozesse von der Ablage oder Änderung dieser Daten zu benachrichtigen (Wikipedia).

nimmt die Rolle eines Dienstanbieters als auch die eines Dienstnutzers an. Die Suche des Dienstnutzers nach einem Dienstanbieters findet jedoch nicht zur Ausführungszeit über ein technisches Dienstverzeichnis statt, sondern wird vom Ersteller einer Widget Orchestrierung (Mashlet) während der Entwicklung bestimmt. Das Dienstverzeichnis steht stellvertretend für eine Suchmaschine (z.B. Google oder Portale mit Widget Sammlungen), die der Entwickler verwendet um passende Dienstanbieter (Widgets) zu finden.

Eine Zusammenstellung von Widgets operiert in einem Kontext in denen sie Daten austauschen. Durch den Austausch von Daten finden Wechselwirkungen statt. Der Ersteller der Widget bestimmt welche Daten zwischen welchen Widgets ausgetauscht werden. Durch die gezielte Stimulation über die grafische Benutzerschnittstelle, führen die Wechselwirkungen zu einem gewünschten Anwendungsverhalten.

4.1.1 Kontext-Daten

Die Kontext-Daten beinhalten im Idealfall alle kollaborativen Daten, die für die Lösung der Aufgabe des Entwicklers in einer Mashlet-Anwendung notwendig sind. Die Daten werden vom Server in einem Blackboard-Modell verwaltet und die verwendete Datenstruktur wird anhand der vom End-User entwickelten Mashlet Definition bestimmt.

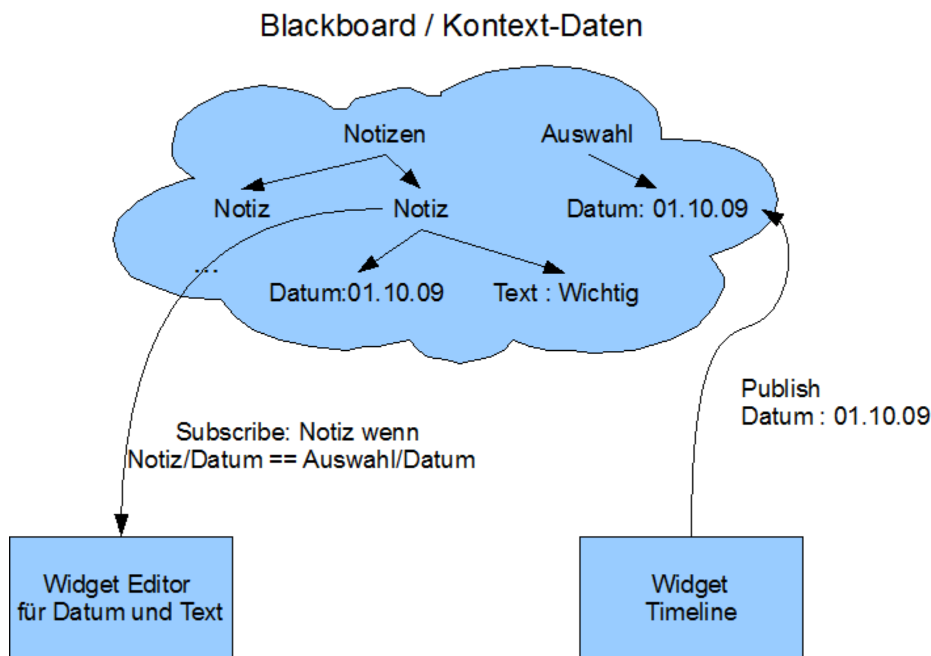

Abbildung 4.2: Zugriff auf die Kontext-Daten

Die hierarchische Datenstruktur des Blackboards wird als XML abgebildet. Gründe für die Nutzung sind:

- Etablierte und weit verbreitete Technologie
  - Hohe Wahrscheinlichkeit der Vertrautheit in der Zielgruppe

    Besseres Verständnis für die Datenstrukturierung bei der Mashlet Gestaltung
  - Gute bestehende Werkzeugunterstützung

    Datenbank gestützte XML-Persistierung
- Anwender lesbare Erscheinungsform: Lesbarer Datenausschnitt für Debugging-Zwecke
- Einsatz als Austauschformat mit anderen Anwendungen

Durch den Einsatz von XML bietet sich die Verwendung von XPath zur gezielten Adressierung einzelner Elemente innerhalb eines XML-Dokuments an. XPath kann als Fundament in der Mashlet Programmierung eingesetzt werden. Genaueres dazu wird in Kapitel 4.1.5 erläutert. Die Vorteile der Nutzung sind:

- Etablierte und weit verbreitete Technologie (Hohe Wahrscheinlichkeit der Vertrautheit in der Zielgruppe)
  - Potentielle Chance das Entwickler diese bereits kennen
  - Vielzahl an Dokumentationen und Tutorials
- Erlaubt jedes Element im XML-Dokument zu adressieren
- Erlaubt komplexe Adressierungen => Bedingte und berechnete Anweisungen, sowie Referenzen sind möglich

Dies bietet eine hohe Flexibilität und somit einen großen Realisierungsspielraum für den End-User. Die Nutzung von XML und XPath unterstützt die Anforderung „Geringer Lernumfang“ insofern, dass bereits vertraute Entwickler keine neue Technologie erlernen müssen und der Anforderung „Fehleranfälligkeit“ dadurch, dass demotivierende Ereignisse beim Erlernen entfallen.

Kontext-Daten, die dynamisch generiert werden, dürfen für nur einen bestimmten Zeitraum (Scope) verfügbar sein. Sicherheitsrelevante Daten dürfen nur für die Dauer einer Transaktion existieren und Daten von z.B. einem Terminplaner müssen permanent vorhanden sein. Deshalb müssen folgende Scopes berücksichtigt werden:

- Transient - Daten werden nur für eine Operation bzw. Transaktion benötigt (z.B. Währungsumrechner)

- Session - Daten werden für die gesamte Dauer einer Sitzung benötigt (z.B. E-Mail Sitzung / Login-Daten)
- Permanent - Daten müssen immer verfügbar sein (z.B. in Statistiken)

Die Nutzung der Scopes erfordert die Handhabung zweier Grenzfälle.

1. Überschreiben eines Elements in unterschiedlichen Scopes - Widget 1 schreibt ein permanentes Element und welches später von Widget 2 durch ein transientes Element überschrieben wird. Bleibt das permanente Element bestehen ?
2. Schreiben eines Elements in unterschiedlichen Scopes, z.B. wenn es vorher gelöscht wurde.

Die beiden Grenzfälle lassen sich vermeiden, wenn bei der Erstellung oder spätestens bei der Ausführung des Wirings überprüft wird, ob überlappende Adressierungen in unterschiedlichen Scopes existieren und als Fehler bezeichnet werden.

Bei der Handhabung der Kontext-Daten zur Laufzeit werden zwei Sichten unterschieden:

- Widget Sicht
  - Aktivität Schreiben
  - Aktivität Lesen
- Server Sicht
  - Daten Verwaltung
  - Daten Persistierung
  - Daten Wiederherstellung

Widget Sicht

Jedes Widget hat Zugriff auf die Kontext-Daten und wer welche Daten hineinschreibt oder liest, wird vom Blackboard über ein Publish / Subscribe Mechanismus bestimmt. Ein Publish ist die schreibende Aktivität neuer Daten und findet durch die Nutzung eines Out-Ports statt, der die Daten direkt an den Server übermittelt. Das Widget hat keinerlei Kontrolle über die Weiterverarbeitung. Der Server überträgt die empfangen Daten in das Blackboard und ermittelt die Subscriber jeweils anhand der Mashlet Definition. Die Benachrichtigung eines Subscribers ist die lesende Aktivität. Ein Subscribe kann mehrere Datenelemente umfassen und wird immer vollständig bedient, selbst wenn ein Publish nur einen Teil modifiziert (siehe Abb. 4.2). Die Daten werden über den In-Port an das Widget überreicht, der auch die zu bedienenden Datenelemente bestimmt.

Server Sicht

Dem Server obliegt die Verantwortung der Verwaltung der Kontext-Daten. Daten müssen persistent gehalten werden, um eine Unterbrechung und eine spätere Wiederaufnahme der Verarbeitung zu ermöglichen. Einige Widgets besitzen die Möglichkeit ihren internen Zustand zu persistieren, aber dieses Verhalten ist nicht für alle garantiert. Desweiteren entstehen zur Laufzeit kontextrelevante Daten, die nicht alleine in den Zuständigkeitsbereich eines einzelnen Widgets gehören, weshalb das System eine Persistierungsmöglichkeit bereitstellen muss.

Die Wiederherstellung der persistierten Daten nutzt das Publish / Subscribe Verfahren für die Synchronisation der Widgets aus. Dazu wird zuerst das XML-Dokument vollständig in den Speicher geladen und anschließend werden alle Wirings sukzessive überprüft. Als erstes wird das Attribute „selectPath“ ermittelt und Elemente selektiert. Das Ergebnis enthält entweder eine Menge von Treffern oder sie ist leer. Jeder Treffer kann mehrere Kindelemente besitzen, die mit einen oder mehreren In-Ports korrespondieren. Es wird für jeden Treffer einmal der In-Port aufgerufen und das für jedes interessierte Widget.

Beispiel für eine Benutzerverwaltung:

```
1 <user>
2   <id>Hans</id>
3   <password>hashPW</password>
4 </user>
5 <user>
6   <id>Werner</id>
7   <password>hash123</password>
8 </user>
```

Listing 4.1: Datenmodell nach Ladevorgang

Ein Widget interessiert sich für alle „user“ Einträge und den beiden Elementen „id“ und „password“. Im Mashlet ist ein entsprechendes Subscribe definiert. Das Beispiel Datenmodell enthält insgesamt zwei „user“ Einträge. Der In-Port wird für jeden Eintrag, in diesem Fall zwei mal, benachrichtigt. Der In-Port verlangt zwei Parameter die jeweils mit den beiden Kindelemente „id“ und „password“ des Elements „user“ befüllt werden. Also wird der InPort nacheinander mit den Tupeln $\{Hans, hashPW\}$ und $\{Werner, hash123\}$ aufgerufen und der interne Widget Zustand ist mit den serverseitigen Kontext-Daten synchronisiert und der normale Ablauf kann stattfinden.

4.1.2 Widget

Das Widget ist eine Web-Anwendung, die typischerweise auf einem Webbrowser ausgeführt wird. Das Widget erlaubt über eine grafische Benutzerschnittstelle mit dem Anwender zu intera-

gieren und anwendungsspezifische Aktivitäten auszuführen. Das Aussehen und der Code kann in einer beliebigen auf dem Webbrowser ausführbaren Sprache (HTML, Javascript, ...) implementiert sein. Widgets, die in einer Widget Orchestrierung eingesetzt werden sollen, benötigen eine öffentliche Schnittstelle zur Inter-Widget-Kommunikation. Aufgrund der vielen Widget Spezifikation und vielen im Internet erhältlichen Widgets wird die hier entwickelte Spezifikation auch die Funktion eines Widget Adapters aufweisen. Bestehende Widgets sollen ohne eine Neuimplementierung verwendet werden können. Die Integration erfordert unter Umständen eine besondere Anpassung (Initialisierung, Datentransformationm, ...) die implementiert werden muss. Die Abbildung 4.3 stellt die wesentlichen Komponenten als Schaubild dar.

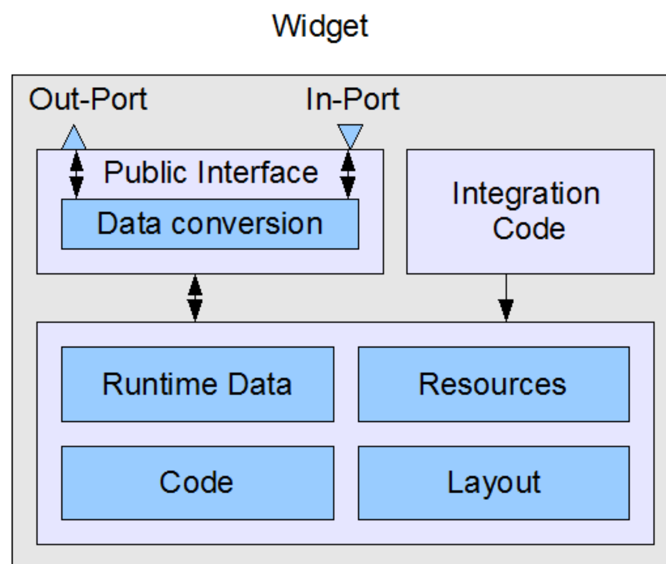

Abbildung 4.3: Komponenten eines Widgets

Aufbauend auf den Erkenntnissen aus Kapitel 3.4.1 wird hier nun eine XML basierte Widget Definition vorgestellt. Ein Widget besteht aus den Komponente (siehe auch Abb. 4.3):

- Öffentliche Schnittstelle
- Integrations Code
- Ressourcen
- Code
- Layout

Die Laufzeit-Daten werden dynamisch erstellt und werden deshalb nicht statisch beschrieben. Die *öffentliche Schnittstelle* dient dazu mit der Außenwelt und im besonderen mit anderen Widgets zu interagieren. Die Interaktion findet im Form von Datenaustausch statt, der durch

ein eingehenden Kanal (In-Port) bzw. ausgehenden Kanal (Out-Port) transportiert wird. Möchte ein Widget A Daten an Widget B verschicken so durchläuft die Nachricht Out-Port A und wird über In-Port B empfangen. Die Daten sind typisiert und können jeweils einen der folgenden Datentypen repräsentieren:

- *string* - Beliebig lange Zeichenkette
- *numeric* - Beliebig lange vorzeichenbehaftete Fließkommazahl
- *date* - Datum gezählt in Millisekunden beginnend vom 01.01.1970
- *time* - Uhrzeit gezählt in Millisekunden
- *dateTime* - Datum und Uhrzeit gezählt in Millisekunden beginnend vom 01.01.1970

Da die Sprache der Widget Implementierung nicht vorgegeben ist, müssen bei Bedarf Typkonvertierungen stattfinden, bevor die Daten weitergeleitet werden. Die Typkonvertierung findet unmittelbar vor dem Versand oder nach dem Empfang einer Nachricht statt und ist in Javascript implementiert. Das Mapping der Daten der Ports auf die internen Widget Daten wird ebenfalls in Javascript implementiert.

Die Komponente *Integrations Code* erlaubt bestehende und externe Widgets zu verwenden, die einer besonderen Anpassung bedürfen. So lassen sich dort spezielle Initialisierungsroutinen oder erweiterte Funktionalitäten unterbringen, die typischerweiser in der gleichen Sprache implementiert sind, wie der Widget Code selbst.

Die *Ressourcen* benennen alle verwendeten ausgelagerten Ressourcen wie Javascript Bibliotheken oder Cascading Style Sheets, die über einen Uniform Resource Identifier[33] (URI) erreichbar sind.

Der *Code* beinhaltet die Widget Logik und kann entweder über eine URI referenziert oder direkt im XML-Dokument eingebettet werden. Er besitzt Einstiegspunkte in Form von Methoden die im Layout verwendet werden, um die Verbindung zwischen Layout und Code herzustellen.

Das *Layout* beschreibt das Aussehen des Widget und kann entweder über eine URI referenziert oder direkt im XML-Dokument eingebettet werden. Das Format ist typischerweise HTML, jedoch sind andere wie Flash und Silverlight ebenfalls möglich. Das Layout verwendet die Einstiegspunkt in der *Code* Komponente um diese lebendig zu gestalten. Es ist auch möglich den Code vollständig im Layout unterzubringen.

Die Implementierung eines Widgets erfordert nicht zwangsweise die Nutzung jeder Komponente, denn Flash Anwendungen können bereits die Ressourcen, den Code und das Layout beinhalten. Eine vollständige Trennung von Code und Layout ist nicht möglich, da die Logik

[33]Ein Uniform Resource Identifier (URI) ist ein Identifikator, die zur Identifizierung einer abstrakten oder physischen Ressource dient.

Teil der GUI ist. Für die Adaption externer Widgets, bietet sich die Verwendung von Referenzen über URIs an. Dadurch reduziert sich der Aufwand der Implementierung und der Größe des XML-Dokuments. Desweiteren finden Widget Aktualisierungen transparent vonstatten die jedoch den *Integrations Code* und die Implementierung der *öffentlichen Schnittstelle* inkompatibel werden lassen kann.

4.1.3 Mashlet

Das Mashlet bzw. die Widget Orchestrierung beschreibt die verwendeten Widgets, wie diese miteinander interagieren, wie diese angeordnet sind und wie die öffentliche Schnittstelle bei einer Wiederverwendung in einem anderen Mashlet aussieht. Es besteht somit aus den vier Komponenten:

- Widget Repository - Liste der verwendeten Widgets
- Wiring - Publish / Subscription der Widgets
- Export - Öffentliche Schnittstelle
- Layout - Anordnung der Widgets

Das *Widget Repository* listet alle Widget Instanzen und der verwendeten Widget Typen auf. Ein Widget Typ ist eine Referenz auf eine Widget Definition wie sie in 4.1.2 erläutert ist und wird über einen eindeutigen Namen im Widget Repository der Laufzeitumgebung oder über eine URI identifiziert. Eine Widget Instanz wird von einem Widget Typ mit Parametern, sofern definiert, und einem im Mashlet eindeutigen Namen instanziiert. Die erlaubten Parameter sind im Widget Typ beschrieben und erlauben pro Widget Instanz das Initialverhalten zu bestimmen.

Der Name der Instanz wird für das *Wiring* verwendet, wo alle Subscription und Publish Aufträge pro Widget Instanz beim Blackboard angegeben werden. Das Blackboard verwendet eine hierarchische Datenstruktur und wird als XML abgebildet. Das Topic[34] entspricht einem Element im XML Dokument und wird über XPath adressiert. XPath Ausdrücke, die sich semantisch überlappen stellen eine Beziehung zwischen Widgets dar. Dies bedeutet, wenn ein Widget publiziert, ein Widget abonniert und deren XPath Ausdrücke übereinstimmende Datenelemente adressieren, dann stehen sie zueinander in Beziehung.

Die Komponenten *Export* beschreibt die öffentliche Schnittstelle, die den Einsatz als Widget in einem anderen Mashlet erlaubt. Sie beinhaltet eine Menge von In- und Out-Ports, vergleichbar mit denen eines Widgets, nur mit dem Unterschied das die Implementierung in Javascript entfällt. Die Implementierung folgt dem Schema des Wirings, wo Datenelemente über XPath Ausdrücke im Blackboard gelesen oder geschrieben werden. Jeder Parameter eines Ports verweist

[34] Ein Thema das abonniert oder publiziert werden kann

auf ein Element. Ein Port kann gezielt einen Knoten selektieren, den die Parameter als den aktuellen Knoten zur weiteren Traversierung verwenden. Dadurch können mehrere Elemente gleichzeitig im Blackboard verändert werden, bevor eine Benachrichtigung an alle Abonnenten verschickt wird.

Das *Layout* ist ein HTML Dokument, auf dem die Position aller Widget Instanzen definiert werden. Jede Instanz wird mit ihrem Namen adressiert und so einer Position zugeordnet.

4.1.4 Inter-Widget-Kommunikation

Die Kommunikation zwischen den Widgets findet indirekt und reaktiv über das Blackboard statt. Ein Publish kann aktiv von einem Widget ausgelöst werden und überträgt die Daten auf das Blackboard, aber ein Subscriber reagiert immer nur auf ein Publish. Dies hat den Vorteil der losen Koppelung, wodurch Widgets leicht ausgetauscht werden können. Desweiteren kann über das Blackboard mehr Daten geliefert werden, als ein Widget eigenständig bereitstellen kann, denn ein In-Port kann mehr Parameter aufweisen als ein Out-Port. Eine Nachricht beinhaltet eine Menge von typisierten Datenelementen die durch einen Kontextbezeichner interpretiert werden. Die Struktur ähnelt der einer Methode in funktionalen Programmiersprachen.

4.1.5 Wiring

Das Wiring erlaubt Widgets reaktiv miteinander über die auf dem Server in XML abgebildeten Kontext-Daten (Blackboard) zu interagieren. Die Zuordnung findet über ein Publish/Subscribe Mechanismus statt, das mit XPath formuliert wird. XPath erlaubt die gezielte Selektion innerhalb eines XML-Dokuments und somit der Kontext-Daten. Alle Daten die ein Widget liefert, werden immer in das Blackboard transferiert und alle Daten die es benötigt, werden daraus gelesen (siehe Abb. 4.2). Diese Vorgehensweise erlaubt es mit Widgets zu interagieren, die mehr Parameter aufweisen als ein anderes Widget bedienen kann. Die fehlenden Daten werden immer aus dem Blackboard gelesen.

Nicht immer ist ein aufwendiges Widget nötig, um ein Ereignis anzustoßen. Zum Beispiel das Aktualisieren einiger Daten, die vom Benutzer angestoßen werden sollen. HTML bietet dazu eine Vielzahl von brauchbaren GUI-Elementen, die man für einfache Tätigkeiten nutzen könnte. Widgets, die solche Ereignisse auslösen, brauchen keine eigenständige Daten zu produzieren oder zu konsumieren. Es reicht wenn sie den Inhalt des Blackboards verändern können, um eine Kettenreaktion auszulösen. Dazu werden Out-Ports ohne Parameter verwendet, die in der Orchestrierung mit Referenzen auf das Blackboard genutzt werden. So könnte ein „Knopf“ Widget den Out-Port „Update“ besitzen, der in der Orchestrierung den Inhalt eines Text-Widgets in einen vorher selektierten Termin überträgt.

Jedes Mashlet besitzt ein XML-Dokument, dass von allen Widget-Instanzen als ein gemeinsames Datenmodell (Blackboard) genutzt wird. Die XML-Spezifikation erlaubt und fordert nur ein Wurzelelement, weshalb intern das Wurzelelement fest auf „model“ verdrahtet wird. Alle XPath-Ausdrücke im Wiring werden für eine leichtere Bedienung ohne dieses Wurzelelement verwendet. Der Ausdruck „/“ wird intern auf „/model/“ abgebildet. Der Ausdruck „.“ ist im „selectPath“ nicht erlaubt, da dieser für die Selektion des initialen Knotens zuständig ist. Anders ausgedrückt: Die Selektion des aktuellen Knotens eines nicht vorhandenen aktuellen Knoten führt zum Fehler. Alle anderen XPath-Ausdrücke sind erlaubt und funktionieren wie in der XPath-Spezifikation angegeben, mit drei Ausnahmen:

1. Processing Instructions und Kommentare dürfen nicht adressiert werden
2. Nicht existente Elemente werden bei der Traversierung erstellt. Bei bedingten Adressierungen werden relevante Elemente mit entsprechenden Werten gefüllt (z.B. „/el[v=/a]“).
3. Konstante Werte werden immer als „String“ betrachtet.
4. Single Select - Es wird immer der erste Treffer verwendet

Publish/Subscribe

Die Adressierung von atomaren Daten im Blackboard geschieht durch die Angabe von XPath-Ausdrücken. Ein Publisher kann über *Out-Ports* folgende drei Aktionen ausführen:

- Aktualisieren - Das Setzen eines neuen Wertes bei einem bereits existierenden Knoten
- Schreiben - Identisch mit der Aktion aktualisieren, jedoch wird der Zielknoten mit allen seinen Elternknoten erstellt, wenn diese fehlen
- Löschen - Das Entfernen eines existierenden Knotens und all seinen Elternknoten, sofern diese keine weiteren Kindelemente aufweisen

Ein Subscriber wird über *In-Ports* definiert. Publisher und Subscriber stehen in Beziehung zueinander, sobald deren über XPath selektierte Knoten sich überlappen. Sendet das Publisher Widget Daten an den Server, entscheidet der Server anhand der Orchestrierungsdaten wo die Daten ins Blackboard geschrieben werden. Anschließend überprüft es anhand der XPath-Ausdrücke in In-Ports, welche Subscriber an den veränderten Werten interessiert sind. Es stellt alle Daten zusammen, die das Wiring für den In-Port vorgibt und benachrichtigt das jeweile Widget über die Änderung (siehe Abb. 4.4). Wird ein Subscriber über ein Ereignis informiert, so ist es, abhängig von der Widget Implementierung, in der Lage eine Publish-Aktion auszulösen. Diese wird vom Server nach der vollständigen Benachrichtigung aller Subscriber verarbeitet.

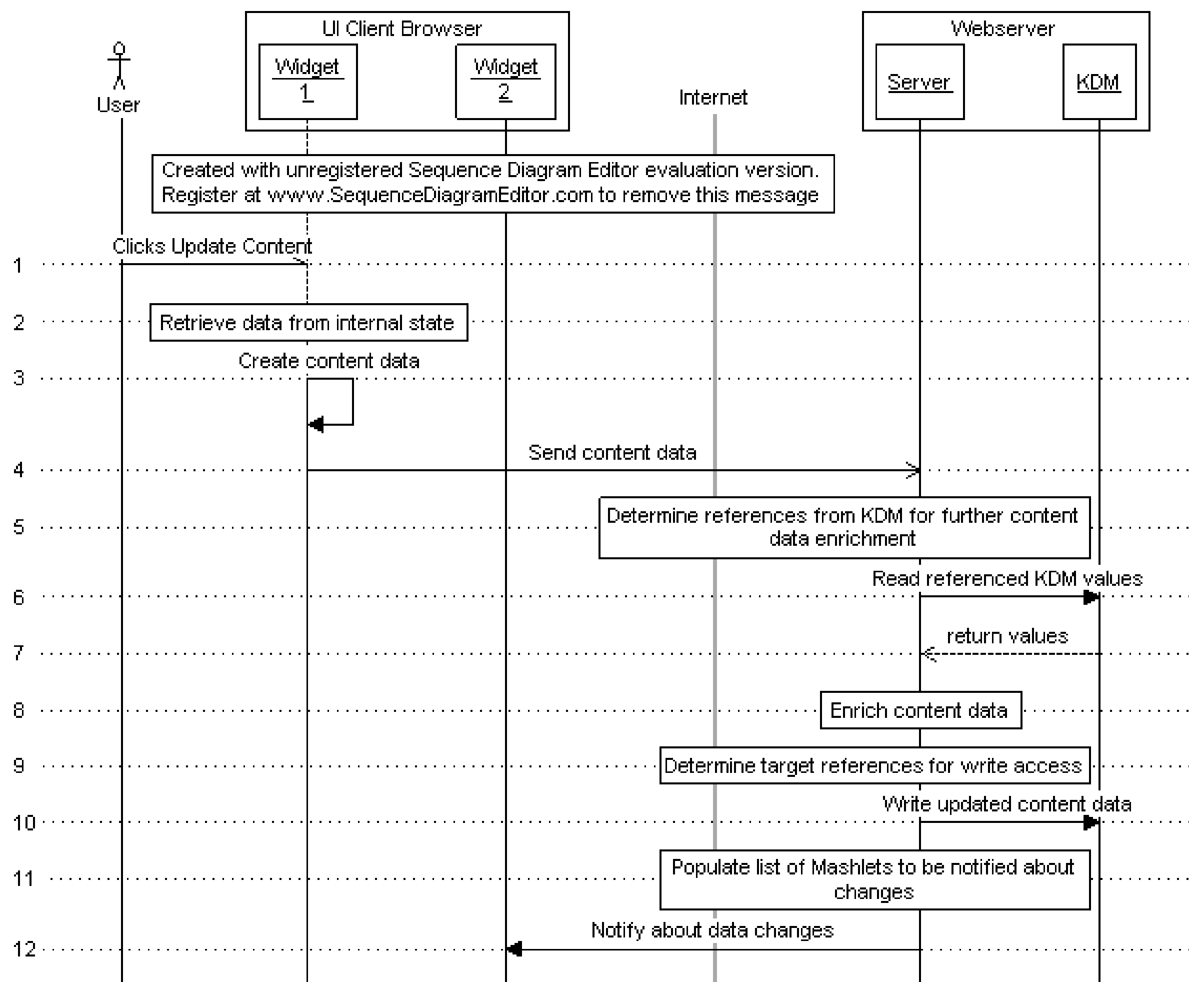

Abbildung 4.4: Sequenz Diagramm: Verarbeitungsschritte

4.2 Abschluss

In diesem Kapitel wurde die allgemeine Architektur der Systemlösung, ihrer einzelnen Bestandteile und das Verhalten bei der Ausführung vorgestellt, die als Grundlage für die Realisierung des Prototypen dient.

Die Verwendung des Blackboard-Modells und einer einheitlichen Widget Definition erlauben den Austausch von Widgets im schnelllebigen Internet mit geringem Aufwand. Die Nutzung von XPath als Adressierungsmechanismus eines Topics im Wiring bietet dem Entwickler eine hohe Flexibilität in den Gestaltungsmöglichkeiten.

5 Realisierung

In diesem Kapitel wird die Realisierung der Widget und Mashlet Beschreibung sowie die Laufzeitumgebung zur Ausführung eines Mashlets vorgestellt.

5.1 Widget

Ein Widget und dessen Komponenten die in Abbildung 4.3 zu sehen sind, werden in XML beschrieben. Hier wird zusammen mit einem Beispiel erläutert wie die Komponenten in XML abgebildet werden.

Die *öffentliche Schnittstelle* wird durch seine In- und Out-Ports definiert. Der In-Port empfängt Daten von außerhalb, kann beliebig viele Parameter aufweisen und wird eindeutig über einen Namen identifiziert. Parameter sind typisiert und werden innerhalb eines In-Ports eindeutig über einen Namen identifiziert (Listing 5.1). Für die Handhabung der Parameter muss unterschieden werden zwischen lokal gültigen Datentypen die in Widgets vorkommen können und den normalisierten System-Datentypen eines Mashlets. Die Unterscheidung zwischen lokalen und System-Datentypen ist für das reibungslose Zusammenspiel der Widgets unabdingbar. Jedes Widget kann potentiell von verschiedenen Entwicklern mit verschiedenen Werkzeugen erstellt worden sein. Dies hat zur Folge, dass die zugrunde liegenden Datenstrukturen und -formate nicht kompatibel untereinander sein müssen. Zur Verdeutlichung sei als Beispiel der Zeitstempel gewählt. Ein Widget könnte diesen in Sekunden beginnend vom 01.01.1970 zählen und ein anderes könnte es als Zeichenkette direkt mit „14:11:00 24.02.2008“ notieren. Um eine reibungslose Interoperabilität zu gewährleisten, muss jeder die gleiche Sprache sprechen. Das Mapping und die Normalisierung der lokalen Datentypen auf die System-Datentypen findet in der Komponente „öffentliche Schnittstelle“ statt und unterliegt der Verantwortung des Widget Erstellers. Der Bestand der zu unterstützenden System-Datentypen sind in Kapitel 4.1.2 zu entnehmen.

```
1  <inPort name="editor">
2    <element name="text" type="string" />
3    <impl>
4      <js>this.yahooEditor.setEditorHTML(text);</js>
5    </impl>
```

```
6   </inPort>
7   <outPort name="editor">
8     <element name="text" type="string">
9       <impl>
10        <js>this.yahooEditor.saveHTML();
11      return this.yahooEditor.get('element').value;</js>
12      </impl>
13    </element>
14  </outPort>
```

Listing 5.1: Beispiel Widget: Öffentliche Schnittstelle (In/Out-Ports)

Für jeden In-Port gibt es einen Implementierungsabschnitt, der alle Daten eines In-Ports in kompatibler Form an die interne Widget Datenverarbeitung weiterleitet. Ihr obliegt weiterhin die Verantwortung invalide Wertepaare zu verwerfen, die z.B. in Form von leeren Werten übergeben werden. Das Schema unterstützt gegenwärtig nur eine Javascript Implementierung, jedoch kann es leicht durch andere Sprachen ergänzt werden, indem weitere Elemente neben der von Javascript aufgeführt werden. Die Javascript Implementierung sieht vor, dass der Widget Ersteller einen Methodenrumpf verwendet, in dem jeder Parameter über eine Variable mit dem Namen des Parameters zur Verfügung steht.

Der Out-Port verschickt Daten die das Widget produziert nach außen und kann wie der In-Port beliebig viele Parameter aufweisen. Der Parameter eines Out-Ports weist die gleichen Eigenschaften die des In-Port Parameters auf, jedoch besitzt es zusätzlich einen Implementierungsabschnitt. Jeder Parameter wird jeweils einzeln über eine Implementierung ermittelt, der das Datum aus dem Widget-Datenpool ermittelt und die Datennormalisierung vornimmt. Die Javascript Implementierung sieht vor, das der Widget Ersteller nur einen Methodenrumpf implementiert und den normalisierten Wert des Parameters als Rückgabewert übergibt.

Die *Ressourcen* werden als Referenzen im Format URI angegeben und unterstützt werden die Javascript Bibliotheken und Cascading Style Sheets (CSS) (Listing 5.2).

```
1     <references>
2       <css>http://yui.yahooapis.com/2.8.0r4/build/assets/skins/sam/skin.css
          </css>
3       <js>http://yui.yahooapis.com/2.8.0r4/build/yahoo-dom-event/
          yahoo-dom-event.js</js>
4       <js>http://yui.yahooapis.com/2.8.0r4/build/element/element-min.js</js
          >
5       <js>http://yui.yahooapis.com/2.8.0r4/build/container/
          container_core-min.js</js>
6       <js>http://yui.yahooapis.com/2.8.0r4/build/menu/menu-min.js</js>
7       <js>http://yui.yahooapis.com/2.8.0r4/build/button/button-min.js</js>
8       <js>http://yui.yahooapis.com/2.8.0r4/build/editor/editor-min.js</js>
```

```
9    </references>
```

Listing 5.2: Beispiel Widget: Ressourcen

Der *Widget Code* und das *Layout* werden nicht strikt getrennt. Das *Layout* beinhaltet ein vollständiges HTML Dokument inklusive der Widget Logik und kann nur als Referenz im Format URI angegeben werden. Widgets, die über kein expliztes Layout-Dokument verfügen, werden über die Komponente *Integration Code* und optional *Ressourcen* eingebunden. Dies ist bei Widgets, die maßgeblich ihr Layout über Javascript und DOM realisieren notwendig.

Der *Integrations Code* erlaubt etwaigen Widget spezifischen Code zu implementieren und unterstützt gegenwärtig nur *Javascript* (Listing 5.3). So lassen sich Widget interne Initialisierungsroutinen aufrufen bzw. erweitern oder um Methoden zum Aufruf der Out-Ports ergänzen. Der Code kann durch drei Ereignisse ausgeführt werden:

- onInit - Initialisierung des generierten Javascript Code zur Kommunikation zwischen Widget und Server.
- onLoad - Tritt beim Laden der HTML Seite auf.
- onDomReady - Tritt beim erfolgreichen Erstellen des DOM Baumes der HTML Seite auf.

```
1    <script>
2      <js>
3      <onLoad><![CDATA[var config = {
4        dompath : false,
5        animate : true
6      };
7      this.yahooEditor = new YAHOO.widget.SimpleEditor(this.id, config);
8      this.yahooEditor.on('toolbarLoaded', function() {
9        this.toolbar.set("titlebar","${title}");
10       this.toolbar.collapse();
11     }, this.yahooEditor, true);
12
13     var notifyEditorChange = function() { ${outport_editor} };
14     this.yahooEditor.render();
15     this.yahooEditor.on('editorKeyUp',notifyEditorChange);
16     ]]></onLoad>
17     </js>
18   </script>
```

Listing 5.3: Beispiel Widget: Integrations Code

Ein Widget ist bei der Instanziierung konfigurierbar. Die Parameter der Konfiguration sind jederzeit in jedem Implementierungsabschnitt verfügbar (Listing 5.4). So kann in der Komponente

Integrations Code wie auch bei der *öffentlichen Schnittstelle* über Platzhalter als Variable darauf zugegriffen werden (Siehe Listing 5.3 Zeile 9). Die Parameter enthalten eine Beschreibung und einen Standardwert, der von außen überschrieben werden kann.

```
1  <configuration>
2    <value key="title" type="string" />
3  </configuration>
```

Listing 5.4: Beispiel Widget: Konfiguration

5.2 Mashlet

Ein Mashlet bzw. die Widget Orchestrierung und dessen Komponenten die in Kapitel 4.1.3 benannt sind, werden in XML beschrieben. Hier wird zusammen mit dem Beispiel einer Organizers-Anwendung erläutert wie die Komponenten in XML abgebildet werden. Das Beispiel besteht nur aus sechs verschiedenen Widget Instanzen die miteinander so verdrahtet werden, das Termine angelegt, angesehen, geändert und gelöscht werden können. Aufgrund des Umfangs werden nur Ausschnitte des XML-Dokuments vorgestellt.

Das *Widget Repository* benennt alle verwendeten Widgets und instanziiert sie optional mit Parametern entsprechend der Widget Konfiguration (Listing 5.5). Widgets werden anhand einer URI oder eines im System eindeutigen Namen referenziert. Die Instanzen besitzen einen im Mashlet eindeutigen Namen für die Verwendung in der Komponente *Wiring*

```
1  <widget name="YahooEditor" id="widget_editor">
2    <property key="title" value="Termin" />
3  </widget>
4  <widget name="Calendar" id="widget_calendar" />
5  <widget name="Timeline" id="widget_timeline" />
6  <widget name="Button" id="update"/>
7  <widget name="Button" id="new" />
8  <widget name="Button" id="delete" />
```

Listing 5.5: Beispiel Mashlet: Widget Repository

Das *Wiring* beinhaltet für jede Widget Instanz ein Wiring, das benennt welche Topics bzw. Datenelemente abonniert oder publiziert werden. Die Adressierung erfolgt über XPath. Das Kalendar Widget publiziert immer nur das vom Anwender ausgewählte Datum (Listing 5.6).

```
1  <wiring widget="widget_calendar">
2    <outPort name="select">
3      <wire>
4        <element name="date" path="/selectedDate"/>
```

```
5        </wire>
6      </outPort>
7    </wiring>
```

Listing 5.6: Beispiel Mashlet: Editor Wiring

Das Editor Widget im Listing 5.7 hat die Funktion der Darstellung und Änderung eines Termins und abonniert und publiziert deshalb das gleiche Topic. Publiziert das Widget etwas so wird es nicht benachrichtigt, sondern erst sobald eine andere Widget Instanz im gleichen Topic etwas publiziert.

```
1    <wiring widget="widget_editor">
2       <outPort name="editor">
3         <wire>
4           <element name="text" path="/editor"/>
5         </wire>
6       </outPort>
7       <inPort name="editor">
8         <wire>
9           <element name="text" path="/editor"/>
10        </wire>
11      </inPort>
12   </wiring>
```

Listing 5.7: Beispiel Mashlet: Editor Wiring

Das Timeline Widget ist für die übersichtliche Darstellung der Termine und der Auswahl des zu ändernden Termins. Es publiziert den darzustellenden Inhalt für das Editor Widget und reagiert auf den Zeitraum zur Darstellung der Termine vom Widget Kalendar. Das Topic „selectedEntry“ beinhaltet den vom Benutzer ausgewählten Termin der publiziert und abonniert wird. Änderungen am Termin lassen sich darüber realisieren. Weiterhin wird das Topic „entry“ abonniert, das alle bekannten Termine enthält. Dadurch können Termine erstellt, geändert oder gelöscht werden (Listing 5.8).

```
1    <wiring widget="widget_timeline">
2      <outPort name="selectedEntry" >
3        <wire>
4          <element name="text" path="/editor"/>
5        </wire>
6      </outPort>
7      <outPort name="selectedEntry" selectPath="/selectedEntry">
8        ...
9      </outPort>
10     <inPort name="selectedEntry" selectPath="/selectedEntry">
11       ...
```

```
12     </inPort>
13     <inPort name="setEntry" selectPath="/entry">
14       ...
15     </inPort>
16     <inPort name="select">
17       <wire>
18         <element name="date" path="/selectedDate"/>
19       </wire>
20     </inPort>
21   </wiring>
```

Listing 5.8: Beispiel Mashlet: Timeline Wiring

An diesem Beispiel ist zu erkennen, das es mehrere Möglichkeiten gibt, ein funktionsfähiges Wiring zu konstruieren. So könnte z.B. der Editor direkt das Topic „selectedEntry/text“ anstatt des „editor“ abonnieren.

Ist erstmal ein Termin selektiert und im Editor geändert worden so kann die Änderung per Knopfdruck mit dem Widget Timeline synchronisiert werden. Das Widget Update ist nichts weiter als ein Knopf das keinen In-Port und nur einen Out-Port ohne Parameter besitzt. Es ist jedoch möglich, im Wiring beliebig viele Topics zu publizieren welches im Listing 5.9 zu sehen ist. Der Inhalt des Editors wird in den aktuellen Termin kopiert. Der Inhalt ließe sich auch einfacher durch */selectedEntry/text* und *selectedEntry/title* übertragen.

```
1  <wiring widget="update">
2    <outPort name="action">
3      <wire>
4        <path srcpath="/editor" destpath="/entry[date=/selectedEntry/date]/
             text"/>
5        <path srcpath="/editor" destpath="/entry[date=/selectedEntry/date]/
             title"/>
6      </wire>
7    </outPort>
8  </wiring>
```

Listing 5.9: Beispiel Mashlet: Update Wiring

Die Anzahl der erstellten Topics und der In- und Out-Ports der Widgets ist entscheidend für die Gestaltungsmöglichkeiten der Realisierung.

5.3 Laufzeitumgebung

Die prototypisch implementierte Laufzeitumgebung, die für die Ausführung von Mashlets zuständig ist, ist ein in Java implementiertes Servlet[35]. Das Servlet wird mit dem Web Application Framework Wicket[36] entwickelt und läuft in dem Servlet-Container und Web-Server Jetty[37]. Der Prototyp erlaubt nur die Ausführung eines Mashlets, welches hart verdrahtet ist. Das Mashlet ist eine Demo-Anwendung und stellt einen simplen Organizer dar. Die Kopplung des Layouts mit einem Mashlet geschieht über deren eindeutige ID. Die ID wird im HTML-Dokument als ein *id* Attribut in einem HTML-Element hinterlegt. Daran erkennt der Prototyp welches Widget an dieser Stelle platziert wird. Dieser Mechanismus erlaubt die Erkennung von ungebundenen und überflüssigen Widgets.

Die Verwaltung des Blackboards, inklusive der Publisher und Subscriber, wird in der Klasse „AbstractMashlet“ implementiert. Sie wertet die eingegangen Daten der Widgets aus und schreibt sie auf das Blackboard, das als ein im Speicher gehaltenes XML-Dokument abgebildet wird. Bei der Auswertung werden die XPath-Ausdrücke dahingehend in den Wirings der Widgets analysiert, ob es überlappende Adressierungen gibt. Im Falle eines Treffers werden die In-Ports der jeweiligen Widgets (Subscriber) benachrichtigt. Eine Fallunterscheidung der Scopes findet nicht statt. Der Prototyp unterstützt nur den Scope „Session“ und betrachtet alle Angaben auch als solche.

Die Kommunikation zwischen Server und Client wird in der Klasse „AbstractWidgetMarkup“ realisiert. Es erweitert das Mashlet HTML-Dokument, um eine Menge von generiertem Javascript Code, die die clientseitige Kommunikationsfähigkeit gewährleistet. Der Javascript Code handhabt das Verhalten der In- und Out-Ports eines Widgets. Für jeden Out-Port des Widgets wird ein Listener auf dem Server registriert und die URL dazu wird im generiertem Javascript eingebettet. Ein Triggern des Out-Ports veranlasst die Kontaktaufnahme über die entsprechende URL, wobei die normalisierten Parameter des Ports mit übertragen werden. Der Kontakt wird ebenfalls genutzt, um bestehende Subscriber über Änderungen zu informieren. Dazu werden auf dem Server AJAX Ausdrücke generiert und auf dem Client ausgeführt, die dann entsprechende Javascript Methoden der In-Ports aufrufen.

Die Klasse „IFrameWidget“ erweitert die Klasse „AbstractWidgetMarkup“ insofern, das sie die Nutzung von Widgets in IFrame zulässt. Dazu wird die URL des Widgets über einen Redirector Service auf dem Mashlet Server umgeleitet, so das die Seite aus der Sicht des Webbrowsers aus der gleichen Domäne stammt. Dies verhindert die Isolierung des Inhalts und eine Manipulation über Javascript ist möglich.

[35] Als Servlets bezeichnet man Java-Klassen, deren Instanzen innerhalb eines Java-Webservers Anfragen von Clients entgegen nehmen und beantworten (Wikipedia).

[36] `http://wicket.apache.org/`

[37] `http://jetty.codehaus.org/jetty/`

Die Klassen ergeben zusammen mit den Javascript Code-Generatoren im wesentlichen die Laufzeitumgebung. Der größte Aufwand liegt in der Verarbeitung der Wirings, im besonderen der Auswertung der XPath-Ausdrücke, und der Generierung des Javascript Codes als Mediator zwischen Server und dem eigentlichen Widget auf dem Webbrowser auf der Client Seite.

5.4 Wiring Guidelines

Dieser Abschnitt stellt alle Grundaktivitäten vor und wie diese durch das Verdrahten von Datenstrukturen anhand trivial Beispielen realisiert werden können. Folgende Grundaktivitäten sind möglich:

- Element erstellen
- Element auslesen
- Element ändern
- Element löschen
- Element in einer Kollektion hinzufügen
- Element in einer Kollektion auslesen
- Element in einer Kollektion ändern
- Element in einer Kollektion löschen

Jede Aktivität kann jeweils mit direkten oder indirekten Werten arbeiten. Direkte Werte stammen von Widgets und indirekte sind Referenzen auf die Kontext-Daten. Die Nutzung von Referenzen ist erst dann sinnvoll, wenn zuvor plausible Werte direkt in das Blackboard geschrieben wurden. Alle schreibende Aktivitäten werden ausgeführt, bevor interessierte Widgets benachrichtigt werden. Widgets bekunden ihr Interesse über eine Benachrichtigung, indem mindestens ein In-Port auf die Kontext-Daten verweist. Die Adressierung kann auf ein Element verweisen, welches aktuell noch nicht existiert aber irgendwann beschrieben werden kann.

Die Aktivitäten werden anhand von trivial Beispiel erläutert und bestehen aus den XML-Dateien für die Widget Definitionen und den Wirings eines Pseudo Mashlets. Die Beispiele werden dabei auf das Notwendigste reduziert, mit der Folge, das die konkrete Einbettung der externen Widgets und Javascript-Implementierungen nur teilweise aufgeführt wird.

Element erstellen
Es gibt zwei Arten, um ein Element zu erzeugen. Die Unterschiedung liegt bei der Herkunft des Wertes. Entweder liefert das Widget den Wert oder ein bestehender Wert wird über die Kontext-Daten herangezogen. Taugliches Minimalbeispiel für beide Variationen:

```
1 <widget name="trivial_widget">
2   <description>Minimal Beispiel</description>
3   <outPort name="inhalt">
4     <element name="text" type="string">
5       <impl>
6         <js>return "Inhalt"</js>
7       </impl>
8     </element>
9   </outPort>
10 </widget>
```

Listing 5.10: Widget-Definition: Element erstellen

Für die beiden Varianten gibt es jeweils ein Wiring. In beiden Fällen wird das Element „meinInhalt“ erzeugt. Widget liefert einen Wert:

```
1   <wiring widget="trivial_widget">
2     <outPort name="inhalt">
3       <wire>
4         <element name="text" path="/meinInhalt"/>
5       </wire>
6     </outPort>
7   </wiring>
```

Listing 5.11: Wiring: Element erstellen; Widget liefert Wert

Referenzierung der Kontext-Daten:

```
1   <wiring widget="trivial_widget">
2     <outPort name="inhalt">
3       <wire>
4         <path srcpath="/quelle" destpath="/meinInhalt"/>
5       </wire>
6     </outPort>
7   </wiring>
```

Listing 5.12: Wiring: Element erstellen; Referenzierung des Datenmodells

Man beachte, dass es nicht notwendig ist, eine andere Widget-Definition zu verwenden. Das Element „text“ wird einfach ignoriert.

Element auslesen

Elemente werden ausgelesen um ein Widget mit Daten zu beliefern oder wenn ein Element über eine Referenzierung beschrieben werden soll. Die Nutzung der Referenzierung ist im Listing 5.12 Zeile 4 zu sehen. Das Element „quelle“ wird ausgelesen um ihn in Element „meinInhalt“ zu schreiben. Die andere Variante arbeitet über die Verwendung von In-Ports. Minimalbeispiel:

```
1 <widget name="trivial_widget">
2   <description>Minimal Beispiel</description>
3   <inPort name="inhalt">
4     <element name="text" type="string" />
5     <impl>
6       <js>alert(text)</js>
7     </impl>
8   </inPort>
9 </widget>
```

Listing 5.13: Widget-Definition: Element über In-Port auslesen

Das Listing 5.13 zeigt das minimal Beispiel einer Widget-Definition um ein Element der Kontext-Daten auszulesen. Dieses Beispiel öffnet ein Informationsdialog mit dem Inhalt des Wertes. Dabei wird der übergebene Wert auf die Javascript-Variable „text“ abgebildet.

Element ändern

Der Vorgang ist identisch mit der Aktivität „Element erstellen“. Es findet keine Unterscheidung statt.

Element löschen

Der Vorgang ist weitestgehend identisch mit der Aktivität „Element erstellen“. Es finden sich nur minimale Unterschiede. Der Auslöser wird in diesem Beispiel das Widget Button 5.14 sein.

```
1 <widget name="Button">
2   ...
3   <outPort name="select">
4     <element name="date" type="date">
5       <impl>
6         <js>...</js>
7       </impl>
8     </element>
9   </outPort>
10  ...
11 </widget>
```

Listing 5.14: Widget-Definition Kalendar: Selektion des aktuellen Tages

Das Wiring zum Löschen eines Elements sieht folgendermaßen aus:

```
1   <wiring widget="trivial_widget_consumer">
2     <outPort name="delete">
3       <wire>
4         <element path="/delete"/>
5       </wire>
6     </outPort>
```

```
7   </wiring>
```

Listing 5.15: Wiring: Element löschen

Der OutPort „delete“ in 5.15 in Zeile 2 veranlasst das System, das Element der Kontext-Daten zu entfernen und alle Widgets, die an der Adresse lauschen einen leeren Wert zukommen zu lassen. Das Löschen bzw. das übermitteln leerer Daten, wird über die Zeile 4 gekennzeichnet. Dort ist keine Datenquelle (Attribut „name“) angegeben.

Der Konsument besitzt folgende Definition:

```
1 <widget name="trivial_widget_consumer">
2   <description>Minimal Beispiel</description>
3   <inPort name="delete">
4     <element name="value" type="string" />
5     <impl>
6       <js>this.value = value;</js>
7     </impl>
8   </inPort>
9 </widget>
```

Listing 5.16: Widget-Definition: Element löschen

Die Widget-Definition in 5.16 nimmt einen Wert namens „value“ unter dem In-Port „delete“ entgegen. „value“ beinhaltet den aktuellen Wert des zu löschenden Elements, welcher in diesem Beispiel dem Javascript Wert „null“ entspricht. Es wird lediglich eine interne Javascript-Variable auf einen leeren String gesetzt. Das Wiring für das Widget sieht folgendermaßen aus:

```
1   <wiring widget="trivial_widget_consumer">
2     <inPort name="delete">
3       <wire>
4         <element name="value" path="/delete"/>
5       </wire>
6     </outPort>
7   </wiring>
```

Listing 5.17: Wiring: Element löschen

Das Widget lauscht an der Adresse „/delete“ auf Daten. Der Wert wird im Widget nicht verwendet, jedoch ist eine Adresse notwendig, damit das System erkennen kann, wer sich dafür interessiert.

Kollektionen

Zugriffe auf Kollektionen finden immer über direkte Adressierungen statt. Sollte der XPath-Ausdruck mehrere Elemente selektieren, so wird nur das Erste der Kollektion selektiert. Multi-Selektionen sind nicht möglich.

In-Ports von Kollektionen müssen besonderen Anforderungen gerecht werden und folgende Tätigkeiten durchführen können:

1. Neues Element hinzufügen
2. Element selektiv ändern
3. Invalide Elemente ignorieren

Da In-Ports immer nur mit einem Element der Kollektion arbeiten, müssen diese über einen Schlüssel eindeutig zugewiesen werden können. Als Schlüssel dient ein oder mehrere Parameter. Wird der Schlüssel in der internen Kollektion nicht gefunden, so wird ein neues Element hinzugefügt. Ist der Schlüssel jedoch vorhanden, so werden die neuen Parameter übernommen. Ist der Schlüssel ungültig (z.B. es fehlt ein Parameter bei einem zusammengesetzten Schlüssel), so wird das Element ignoriert.

Element in einer Kollektion hinzufügen

Das Wiring ist abhängig von der Implementierung des In-Ports in der Widget-Definition. Der In-Port muss in der Lage sein neue Elemente zu erstellen und bestehende zu ändern.

```
1 <widget name="Consumer">
2   <inPort name="setUser">
3     <element name="user" type="string">
4     <element name="password" type="string">
5     </element>
6       <impl>
7         <js>if(user != null)
8               getOrCreateUser(user).setPassword(password);
9         </js>
10      </impl>
11  </inPort>
12 </widget>
```

Listing 5.18: Widget-Definition: Element erstellen

Als Quelle wird das Producer Widget 5.19 zweckentfremdet.

```
1 <widget name="Producer">
2   <outPort name="produce">
3     <element name="value" type="string">
4       <impl>
5         <js>return "a␣value";</js>
6       </impl>
7     </element>
8   </outPort>
9 </widget>
```

Listing 5.19: Widget-Definition trivial Producer: Generiert Daten

Dieses ist nur in der Lage immer den gleichen Wert zu generieren, welcher als Benutzername und Passwort verwendet wird. In der Realität macht dies keinen Sinn, jedoch dient es nur als ein einfaches Beispiel und kann durch ein sinnvolles Login-Widget ersetzt werden.

```
1  <wiring widget="Producer">
2    <outPort name="produce">
3      <wire>
4        <element name="value" path="/currentUser"/>
5      </wire>
6    </outPort>
7    <outPort name="produce" selectPath="/user[id=/currentUser]">
8      <wire>
9        <element name="value" path="password"/>
10     </wire>
11   </outPort>
12 </wiring>
```

Listing 5.20: Wiring: Element erstellen; Widget liefert Wert

Das Wiring 5.20 veranlasst über den ersten Out-Port in Zeile 2, dass der Wert in „value“ unter „/currentUser“ geschrieben wird. Dieses Element wird als Schlüssel verwendet, um in der Kollektion „/user“ einen Eintrag gezielt zu selektieren. Der zweite Out-Port in Zeile 7 selektiert gezielt einen Benutzer und erstellt diesen bei Bedarf mit dem Kindelement „id“ und dem Wert von „/currentUser“. Weiterhin wird das Kindelement „password“ mit dem Wert in „value“ erzeugt.

Es fehlt nur noch das Wiring des Consumers.

```
1  <wiring widget="Consumer">
2    <inPort name="setUser" selectPath="/user">
3      <wire>
4        <element name="id" path="id"/>
5        <element name="password" path="password"/>
6      </wire>
7    </inPort>
8  </wiring>
```

Listing 5.21: Wiring: Element erstellen; Widget konsumiert

Das Wiring in Listing 5.21 veranlasst das Setzen eines Passwords eines bestimmtes Benutzers in den internen Widget Daten.

Element in einer Kollektion auslesen

Elemente aus einer Kollektion werden über direkte Adressierungen ausgelesen. Das Listing

5.21 kann mit minimaler Veränderung wiederverwendet werden. Es muss lediglich der In-Port in Zeile 2 durch folgendes ersetzt werden:

```
1     <inPort name="setCurrentUser" selectPath="/user[id=/currentUser/id]">
```

Listing 5.22: Wiring: Element erstellen; Widget liefert Wert

Es ist auch möglich Konstanten zu nutzen.

```
1     <inPort name="setAdminPassword" selectPath="/user[id='admin']">
```

Listing 5.23: Wiring: Element erstellen; Widget liefert Wert

Das Listing 5.23 liest Änderungen am Passwort des Benutzers „admin“. Die XML-Element Tag „element“ in Zeile 4 von 5.21 ist überflüssig und kann entfernt werden.

Element in einer Kollektion ändern
Der Vorgang ist identisch mit der Aktivität „Element in einer Kollektion erstellen“. Es findet keine Unterscheidung statt.

Element in einer Kollektion löschen
Das Löschen von Elementen beinhaltet eine Besonderheit. Das Löschen von Daten im internen Zustand eines Widgets muss explizit ausgelöst werden. Der Grund liegt bei Adressierungsart. Es gibt kein vom System vorgegebenes Mapping der XPath-Ausdrücke auf interne Daten eines Widgets und die Benachrichtigung über das Löschen von Elementen geschieht über das übermitteln von leeren Datenmengen. Das Widget kann unmöglich feststellen welches Element in der Kollektion gelöscht wurde. Damit der interne Zustand eines Widgets synchronisiert wird, müssen explizit entsprechende In-Ports verwendet werden, die als Parameter den Schlüssel des Elements in der Kollektion enthält. Das folgende Minimalbeispiel soll dies illustrieren:

Der Auslöser für das Löschen findet über einen Knopf 5.14 statt, welcher folgendermaßen verdrahtet ist:

```
1  <wiring widget="delete">
2    <outPort name="action" selectPath="/entry[date=/selectedEntry/date]">
3      <wire>
4        <element path="text"/>
5        <element path="title"/>
6        <element path="date"/>
7      </wire>
8    </outPort>
9    <outPort name="action" selectPath="/removeEntry">
10     <wire>
11       <element srcpath="/selectedEntry/date" destpath="date" />
12       <element srcpath="/selectedEntry/title" destpath="title" />
13       <element srcpath="/selectedEntry/text" destpath="text" />
14     </wire>
```

```
15      </outPort>
16    </wiring>
```

Listing 5.24: Wiring: Element löschen

Das Wiring 5.24 hat zwei Aufgaben. Erstens wird ein Eintrag der Kontext-Daten über den Out-Port in Zeile 2 entfernt. Es wird explizit ein bestimmtes „entry“ selektiert, wo das Element „date“ mit dem in „/selectedEntry/date“ übereinstimmt. Das Auslassen des Attributs „name“ im XML-Element „element“ kennzeichnet das Löschen. Es ist auch möglich Elemente über eine ungültige Referenzierung[38] zu löschen. Die Widget-Signalisierung findet über den zweiten Out-Port in Zeile 9 statt. Über die drei Parameter ist eine eindeutige Identifizierung des Elements innerhalb des Widgets möglich. Gegebenfalls würde das Datum als eindeutiger Schlüssel genügend.

[38]Ungültig im Sinne von nicht existentes Element

6 Fazit und Ausblick

Diese Arbeit hatte zum Ziel, eine Plattform zur Erstellung und Ausführung von Web-Anwendung bestehend aus verschiedenen Widgets zu entwerfen, mit dessen Hilfe ein weiterer Bereich des „Rapid Web Application Development“ für End-User Development erschlossen werden kann. Zu diesem Zweck wurde auf Basis eines imaginären Anwendungsszenarios ein Softwaresystem entwickelt, das die Erstellung und Ausführung einer Anwendung aus verteilten Widgets ermöglicht.

6.1 Fazit

Die voranschreitende Entwicklung von intelligenten bzw. komplexen Web-Anwendungen und die steigende Akzeptanz und Nutzung dieser lassen darauf schließen, dass ein Bedarf an entsprechenden und im Besonderen benutzerfreundlichen Entwicklerwerkzeuge besteht. Viele Hersteller sehen erfahrene und/oder professionelle Softwareentwickler als ihre Zielgruppe und setzen meist auf Frameworks um „bessere“ Web-Anwendungen erstellen zu können. Diese Frameworks arbeiten meist nur mit professionellen Programmiersprachen, wodurch deren Vorteil den Endbenutzern verwehrt wird. Diese Arbeit soll einen Beitrag dazu leisten, der Endbenutzern erlaubt, anspruchsvolle Web-Anwendungen zu erstellen.

Das in dieser Arbeit entwickelte System erlaubt die Erstellung von Web-Anwendung unter Berücksichtung der manuellen Gestaltung des Layouts. Es erlaubt jedes Widget, welches über Javascript manipulierbar ist, als Baustein zu nutzen. Das Anwendungsverhalten wird über explizite Angaben des Datenaustauschs zwischen Widgets bestimmt. Die Angaben werden im Einklang des Konzepts der domänenspezifischen Sprachen beschrieben. Das System belegt, das die Idee umsetzbar ist, aber bleibt der Frage nach der Akzeptanz schuldig. Dieses müssten über Feldtests beantwortet werden.

6.2 Ausblick

Einige Fragen, die sich durch das entwickelte System ergeben, erfordern weitere Erkenntnisse. Dazu zählen die Erstellungsmöglichkeiten des Layouts und die Nutzung eines grafischen Editors für das Wiring, um End-User eine einfachere Alternative zur Programmierung zu bieten.

Ein weiterer Ausblick ist die Nutzung und Einbettung von Nicht-Web-Anwendungen. Google, Microsoft, Yahoo und andere bieten sogenannte Desktop Gadgets an, die prinzipiell den Widgets ähneln.

Literaturverzeichnis

[Chotirat u. a. 2000] CHOTIRAT, John P. ; PANE, John F. ; RATANAMAHATANA, Chotirat A. ; MYERS, Brad A.: Studying the Language and Structure in Non-Programmers' Solutions to Programming Problems. In: *International Journal of Human-Computer Studies* 54 (2000), S. 237–264. – URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.21.2836&rep=rep1&type=pdf. – Zugriffsdatum: 28.05.2010

[Croquet] : *The Croquet Consortium*. – URL http://croquetconsortium.com/. – Zugriffsdatum: 28.05.2010

[Cypher u. a. 1993] CYPHER, Allen (Hrsg.) ; HALBERT, Daniel C. (Hrsg.) ; KURLANDER, David (Hrsg.) ; LIEBERMAN, Henry (Hrsg.) ; MAULSBY, David (Hrsg.) ; MYERS, Brad A. (Hrsg.) ; TURRANSKY, Alan (Hrsg.): *Watch what I do: programming by demonstration*. Cambridge, MA, USA : MIT Press, 1993. – ISBN 0-262-03213-9

[DOM] W3C: *Document Object Model (DOM)*. – URL http://www.w3.org/DOM/. – Zugriffsdatum: 05.07.2010

[Eclipse] : *Eclipse*. – URL http://www.eclipse.org/. – Zugriffsdatum: 16.08.2010

[EMML] : *EMML - Enterprise Mashup Markup Language*. – URL http://www.openmashup.org/omadocs/v1.0/index.html. – Zugriffsdatum: 24.02.2010

[Endrei u. a. 2004] ENDREI, Mark ; ANG, Jenny ; ARSANJANI, Ali ; CHUA, Sook ; COMTE, Philippe ; KROGDAHL, Pål ; LUO, Min ; NEWLING, Tony: *Patterns: Service-Oriented Architecture and Web Services*. 2004. – URL http://www.redbooks.ibm.com/redbooks/pdfs/sg246303.pdf. – Zugriffsdatum: 16.08.2010

[EUSES] : *EUSES - End Users Shaping Effective Software*. – URL http://eusesconsortium.org/. – Zugriffsdatum: 12.10.2009

[Green und Petre 1996] GREEN, T. R. G. ; PETRE, M.: *Usability Analysis of Visual Programming Environments*. 1996. – URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.47.4836&rep=rep1&type=pdf

[Halbert 1984] HALBERT, Daniel C.: *Programming by example*, Dissertation, 1984. – URL http://www.halwitz.org/halbert/pbe.pdf. – Zugriffsdatum: 24.02.2010

[HCI] : *HCI Bibliography: Human-Computer Interaction Resources*. – URL http://hcibib.org/. – Zugriffsdatum: 16.08.2010

[Hudak 1998] HUDAK, Paul: Modular Domain Specific Languages and Tools. In: *in Proceedings of Fifth International Conference on Software Reuse*, IEEE Computer Society Press, 1998, S. 134–142. – URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.53.5061. – Zugriffsdatum: 24.02.2010

[IBM Mashup Center] : *IBM Mashup Center 2.0*. – URL http://www-01.ibm.com/software/info/mashup-center/. – Zugriffsdatum: 16.08.2010

[Ko und Myers 2004] KO, Andrew J. ; MYERS, Brad A.: Designing the whyline: a debugging interface for asking questions about program behavior. In: *CHI '04: Proceedings of the SIGCHI conference on Human factors in computing systems*. New York, NY, USA : ACM, 2004, S. 151–158. – ISBN 1-58113-702-8

[Ko und Myers 2006] KO, Andrew J. ; MYERS, Brad A.: Barista: An implementation framework for enabling new tools, interaction techniques and views in code editors. In: *CHI '06: Proceedings of the SIGCHI conference on Human Factors in computing systems*. New York, NY, USA : ACM, 2006, S. 387–396. – ISBN 1-59593-372-7

[Ko und Myers 2008] KO, Andrew J. ; MYERS, Brad A.: *Debugging reinvented: asking and answering why and why not questions about program behavior*. 2008. – URL http://faculty.washington.edu/ajko/papers/Ko2008JavaWhyline.pdf. – Zugriffsdatum: 28.05.2010

[LEGO Mindstorm] : *LEGO Mindstorm*. – URL http://mindstorms.lego.com/. – Zugriffsdatum: 16.08.2010

[Lieberman 2001] LIEBERMAN, Henry: *Your wish is my command: programming by example*. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 2001. – URL http://web.media.mit.edu/~lieber/PBE/Your-Wish/. – Zugriffsdatum: 24.02.2010. – ISBN 1-55860-688-2

[Lieberman u. a. 2006] LIEBERMAN, Henry ; PATERNÒ, Fabio ; KLANN, Markus ; WULF, Volker: End-User Development: An Emerging Paradigm. In: LIEBERMAN, Henry (Hrsg.) ; PATERNÒ, Fabio (Hrsg.) ; WULF, Volker (Hrsg.): *End User Development* Bd. 9. Dordrecht : Springer Netherlands, 2006, Kap. 1, S. 1–8. – URL http://dx.doi.org/10.1007/1-4020-5386-X_1. – ISBN 978-1-4020-4220-1

[Myers u. a. 2004] MYERS, Brad A. ; PANE, John F. ; KO, Andy: Natural programming languages and environments. In: *Commun. ACM* 47 (2004), Nr. 9, S. 47–52. – URL http://doi.acm.org/10.1145/1015864.1015888. – Zugriffsdatum: 28.09.2009. – ISSN 0001-0782

[NatProg] : *Natural Programming Project*. – URL http://www.cs.cmu.edu/~NatProg/. – Zugriffsdatum: 28.09.2009

[OpenAjax Hub] : *OpenAjax Hub*. – URL http://www.openajax.org/member/wiki/OpenAjax_Hub_2.0_Specification. – Zugriffsdatum: 16.08.2010

[OpenAjax Widget] : *OpenAjax Metadata 1.0 Widget Spezifikation*. – URL http://www.openajax.org/member/wiki/OpenAjax_Metadata_1.0_Specification_Widget_Metadata. – Zugriffsdatum: 16.08.2010

[OpenMusic] : *OpenMusic*. – URL http://recherche.ircam.fr/equipes/repmus/OpenMusic. – Zugriffsdatum: 16.08.2010

[OpenSocial Gadget] : *OpenSocial Gadget*. – URL http://www.opensocial.org/page/specs-1. – Zugriffsdatum: 16.08.2010

[Ousterhout 1997] OUSTERHOUT, John K.: Scripting: Higher Level Programming for the 21st Century. In: *IEEE Computer* 31 (1997), S. 23–30

[Peltz 2003] PELTZ, Chris: Web Services Orchestration and Choreography. In: *Computer* 36 (2003), Nr. 10, S. 46–52. – ISSN 0018-9162

[REST] : *Wikipedia: Representational State Transfer*. – URL http://de.wikipedia.org/wiki/Representational_State_Transfer. – Zugriffsdatum: 16.08.2010

[Schiffer 1996] SCHIFFER, Stefan: Visuelle Programmierung - Potential und Grenzen. In: *GI Jahrestagung*, URL http://www.schiffer.at/publications/se-96-19/se-96-19.pdf. – Zugriffsdatum: 24.02.2010, 1996, S. 267–286

[Statistisches Bundesamt Deutschland] : *73% der privaten Haushalte haben einen Internetzugang*. – URL http://www.destatis.de/jetspeed/portal/cms/Sites/destatis/Internet/DE/Presse/pm/2009/12/PD09__464__IKT. – Zugriffsdatum: 16.08.2010

[Sutcliffe 2005] SUTCLIFFE, Alistair: Evaluating the costs and benefits of end-user development. In: *WEUSE I: Proceedings of the first workshop on End-user software engineering*. New York, NY, USA : ACM, 2005, S. 1–4. – ISBN 1-59593-131-7

[Sutcliffe u. a. 2003] SUTCLIFFE, Alistair ; LEE, Darren ; MEHANDJIEV, Nik: *Contributions, Costs and Prospects for End-User Development*. 2003. – URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.122.7592&rep=rep1&type=pdf

[W3C Widget] : *Widget Packaging and Configuration*. – URL http://www.w3.org/TR/widgets/. – Zugriffsdatum: 16.08.2010

[Zarandioon und et al. 2008] ZARANDIOON, Saman ; AL. et: *OMOS: A Framework for Secure Communication in Mashup Applications*. 2008

A Inhalt der CD-ROM

Dieser Arbeit ist eine CD-ROM beigelegt. Sie enthält folgende Verzeichnisse:

- **Masterarbeit** beinhaltet diese Arbeit in digitaler Form.
- **Source-Code** enthält den Quelltext des entwickelten Prototyps. Der Quelltext ist in zwei Projekten unterteilt, die in der Entwicklungsumgebung Eclipse (ab Version 3.5) importiert werden können.
- **Distribution** enthält eine ausführbare Fassung des Prototypen die unter Windows über das Batch-File „mashor.bat“ zu starten ist.

Versicherung über Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §22(4) ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 29. August 2010
Ort, Datum

Unterschrift