Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Ausarbeitung AW2

Christian Stachow

Mashup Widgets unter Berücksichtigung von End User Development Konzepten

Fakultät Technik und Informatik
Department Informatik

Inhaltsverzeichnis

Kapitel 1

Einführung

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. Eine Ausprägung ist die Erstellung und Nutzung von Mashups.

Mashup ist ein Begriff aus der Musikbranche, welcher sich mit dem Web 2.0 Hype auch in der Informatik etabliert hat. Es bezeichnet das Verknüpfen von bestehenden Elementen um etwas Neues zu kreieren. Gegenwärtig beschränkt sich das Mashup in der Informatik im wesentlichen auf das Sammeln und Verarbeiten von externen Daten, welche mit einer standardisierten oder individuell angepassten GUI (Widget) dargestellt werden. Die Logik der Datenverarbeitung wird dabei entweder textuell oder mit grafischen Editoren verfasst.

Der Großteil der erstellten Mashups fällt unter die Kategorie Web-Anwendung, da sie auf einen Server laufen und die Benutzer-Interaktion ausschließlich über einen Web-Browser stattfindet. Mashups, die ein aufwendiges Widget besitzen, fallen weiterhin unter die Kategorie Rich Internet Application (RIA). Ein hohes Maß der Programmlogik wird dafür auf den Web-Browser übertragen. Die Erstellung von Mashups kann sich somit in den Bereich „Rapid Web Application Development“ mit einreihen, welche für eine schnelle Entwicklung von Web-Anwendung steht.

Das Konzept der Mashup-Entwicklung besitzt viele Schnittpunkte zu großen Bereichen wie End-User Development (EUD), Rapid Web Application Development, Usability usw. die wiederum untereinander ebenfalls Schnittpunkte aufweisen, weshalb hier aufgrund der Komplexität der Fokus alleine auf EUD liegt. Ziel dieser Ausarbeitung ist die Analyse und Bewertung von End-User Development (EUD) Konzepten zur Vernetzung von Widgets. Dazu werden vier EUD Konzepte und einige ausgewählte Mashup-Entwicklerwerkzeuge, die den Rapid Web Application Development Ansatz unterstützen, vorgestellt, die dann als Adaption auf die Idee des „Mashup Widgets“ bewertet werden.

Kapitel 2

Related Work

2.1 End User Development Konzepte

„End User Development“ (EUD) (Lieberman u. a., 2006) beschreibt den Bereich der Softwareentwicklung für Personen (End-User), deren Hauptaufgabe nicht die Softwareentwicklung ist, die jedoch Domänenwissen aus dem Anwendungsfeld besitzen. Diese „End-User“ oder auch Endbenutzer genannte Gruppe von Personen, steht im Fokus des Themas mit dem Ziel sie zu motivieren und ihnen die Programmierung zu vereinfachen. Die gewonnenen Erkenntnisse kommen auch den professionellen Softwareentwicklern zugute. In diesem Abschnitt werden die vier wesentlichen Konzepte des „End User Developments“ und einige ausgewählte Mashup-Entwicklerwerkzeuge vorgestellt.

2.1.1 Natürliche Programmierung

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 „Natürliche Programmierung“ versteht man als Ziel, 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 eines 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) versucht den Misstand zu beheben, in dem es zukünftige Entwicklungen beeinflusst. Es bietet dafür eine Reihe von benutzerzentrierten Fakten an, die als Leitfaden für Design-Entscheidungen dienen. Dies befähigt Entwickler bewusster über potentielle Probleme zu urteilen und alternative Lösungswege zu beschreiten, die dem Empfinden von Menschen stärker ähneln. Es existieren bereits zahlreiche Projekte und Berichte zu alternativen Debugging-Techniken (Ko und Myers, 2004, 2008), Validierungstechniken und Code/Daten-Visualisierungen, wovon zwei zur besseren Verständlichkeit vorgestellt werden.

Eine Innovation betrifft das Debuggen von Programmen durch das „Why, why not?“ (Ko und Myers, 2004, 2008) Konzept. Bei falschen Programmverhalten fragt man sich häufig warum dies oder warum dies nicht geschieht. Typische Entwicklungsumgebungen erlauben diese Art von Fragestellung nicht. Alice ist eine Entwicklungsumgebung zur Erstellung von

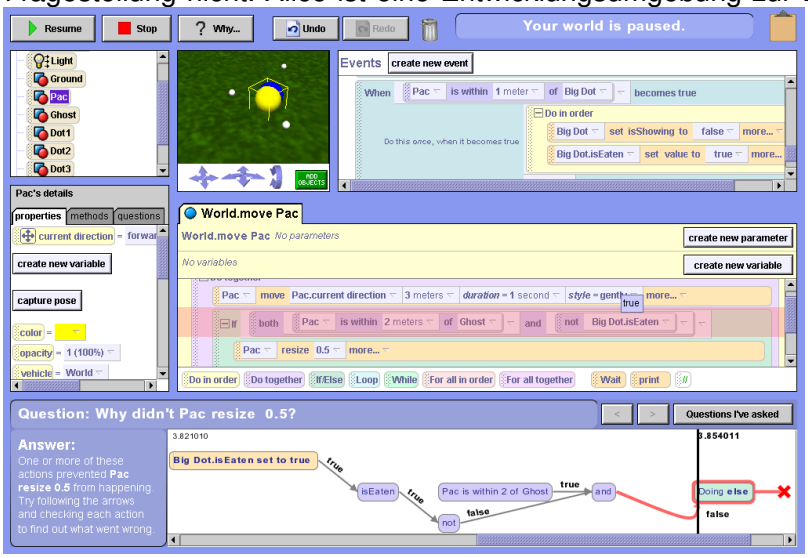

Abbildung 2.1: Alice WhyLine

dreidimensionalen medialen Inhalten, die erstmals das „Why, why not?“ Konzept umsetzt (siehe Abb. 2.1). Dabei wird der mit Meta-Informationen angereicherte Programmfluss grafisch dargestellt. Ursachen von Ereignissen und Eigenschaftsänderungen können anhand von generierten Fragen[1], welche durch statische und Laufzeit Analysen ermittelt werden, aufgedeckt werden. Dabei wird die Ereigniskette, die zu der gestellten Frage führt, grafisch dargestellt, die daraufhin weiter untersucht werden kann.

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[2] wurde der Standard weiter angehoben mit

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

[2] Eclipse ist eine erweiterbare offene Entwicklungsplattform.

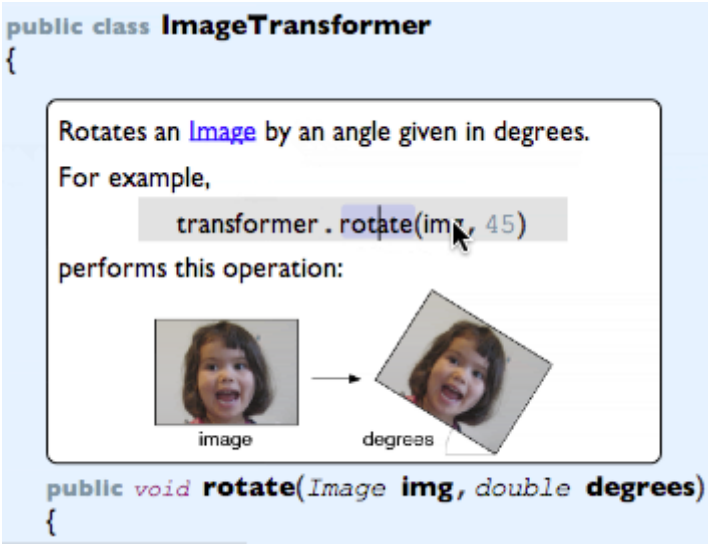

Abbildung 2.2: Barista: Eine Medien angereicherte Annotation einer Java-Methode

Funktionen wie Code-Vervollständigung, Code-Folding[3] und weiteren. Barista (Abb. 2.2)(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. Dies beinhaltet beispielweise das verkleinern von Code-Fragmenten, alternative Darstellungen von Code-Fragmenten (z.B. Mathematische Formeln, boolesche Ausdrücke), In-Code Dokumentation via HTML, etc.

2.1.2 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

[3] Zusammenklappen von Code-Fragmenten

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 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 (Schiffer, 1996) bewertet die visuelle Programmierung anhand von fünf Thesen und bezeichnet den Nutzen als:

> 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.

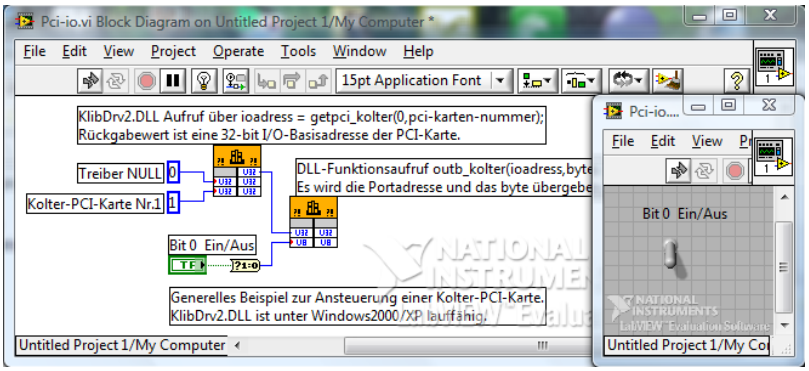

Abbildung 2.3: LabVIEW Datenfluss und Bedienelemente

LabVIEW ist eine beispielhafte Umsetzungen der Programmier-Metapher der visuellen Programmierung die hier kurz vorgestellt wird. Es existieren weitere interessante Anwendungen wie Marten[4] oder PureData[5], auf die jedoch nicht näher eingegangen wird.

LabVIEW wird hauptsächlich von Wissenschaftlern und Ingenieuren zur Datenerfassung und -verarbeitung verwendet. Die Hauptanwendungsgebiete sind die Mess-, Regel- und Automatisierungstechnik. Für die Programmierung in LabVIEW wird ein Datenfluss modelliert. Das Datenmodell beinhaltet Funktionsblöcke (siehe Abbildung 2.3) mit Ein- bzw. Ausgängen,

[4] MacOSX Version einer visuellen Software-Entwicklungsumgebung: Marten

[5] PureData's grafische Programmierung der Verarbeitung von Audio- und Video-Daten

die miteinander verbunden werden können. Bei der Ausführung des Programms, erhält man je nach verwendeten Funktionsblock ein Bedienelement für die Steuerung und Auswertung.

2.1.3 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.

Ein System muss diese beide Funktionen unterstützen, damit es als ein „Programming by Example“ System genannt werden kann. Bestimmte Systeme erlauben das Programmieren in der Benutzerschnittstelle, jedoch mangelt es ihnen an der Möglichkeit Aufzeichnungen der Programmausführung anzustellen. Deshalb fallen sie in eine andere Kategorie. Zum Beispiel unterstützen viele Texteditoren eine gemeinläufige Funktion namens „Makro“, in denen manuell eingegebene Sequenzen von gewöhnlichen Befehlen gespeichert und wiederholt ausgeführt werden können. Ein weiteres Beispiel sind Skripte für Skriptsprachen.

Ein Beispiel für das „Programming by Example“ Paradigmas ist der Text-Editor UltraEdit[6]. Er erweitert die Makro-Funktion, indem es selbstständig eingegebene Benutzerbefehle aufzeichnet und als ein Makro zusammenfasst.

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, ein Programm welches druckt, könnte eine Option für einfachen oder doppelten Zeilenabstand anbieten.

- Andere Systeme lassen den Benutzer Programme erstellen, welche aus Tastenanschläge oder anderen einfachen Benutzeraktionen bestehen. Programmierung mittels Tastenanschläge ist besonders gebräuchlich in Text-Editoren. Die Sequenzen die dort erstellt werden, werden *Makros* genannt. Der Vorteil dieser Programmiertechnik ist,

[6] Homepage: http://www.ultraedit.com/

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 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.

Häufig möchte man jedoch keine exakten Wiederholungen programmieren, sondern die Programme parametrisieren oder generalisieren um eine Wiederverwendbarkeit zu gewährleisten.

2.1.4 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. Bespiele hierfür sind Yacc und Lex, die als syntaktische und lexikalische Analysewerkzeuge für den Compilerbau verwendet werden. 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.

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[7] 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.

2.2 Mashup Entwicklerwerkzeuge

Das Konzept der natürlichen Programmierung betrifft im wesentlichen bereits existierende Anwendungen, weshalb hier, für die spätere Auswertung, einige ausgewählte Mashup Entwicklerwerkzeuge kurz vorgestellt werden. Werkzeuge, die nur für die Generierung von Mashups ohne Rücksicht auf die GUI entwickelt wurden, werden nicht berücksichtigt.

2.2.1 Intel Mash Maker

Intel Mash Maker weicht vom stereotypischen Mashup-Ansatz ab. Es verwendet keine API's zur Datengewinnung sondern betreibt ausschließlich Screen-Scraping[8] der Internet-Seiten die mit Widgets aufgewertet werden sollen. Verfügbare Widgets lassen sich per Drag&Drop verwenden und werden über wenige Eingabefelder konfiguriert. Neues Verhalten, welches durch die bestehenden Widgets nicht abgedeckt werden, muss eigenständig ausprogrammiert werden. Das einbinden von Fremd-Widgets ist möglich.

[7] 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).

[8] Manuell angepasster Algorithmus zur Untersuchung und Extraktion von Informationen einer Internet-Seite

2.2.2 WidgetBox

WidgetBox besitzt eine stark eingeschränkte Funktionsvielfalt. Es spezialisiert sich dabei auf die im Internet häufig eingesetzten Typen von Widgets und stellt für diese entsprechende Templates über Assistenten zur Verfügung. Die Templates müssen nur mit wenigen passenden Parametern besetzt werden, um funktionsfähige Widgets zu erstellen. Das Look&Feel dieser Templates lässt sich begrenzt beeinflussen. Das Theme bzw. die Hintergrundfarbe und einige Darstellungsvarianten sind auswählbar. Falls kein Template passen sollte, so bietet WidgetBox die Möglichkeit an, das Widgets durch Javascript oder Actionscript zu auszuprogrammieren.

2.2.3 WSO_2

WSO_2 ist eine Open-Source Lösung, die für die Mashup Entwicklung vollständig auf Javascript setzt. Sowohl die Logik der Mashup Komponenten als auch die GUI, das Widget, wird in Javascript realisiert. Für Webentwickler ist der starke Fokus auf Javascript ein Vorteil, denn sie sind bei ihrer Arbeit in irgendeiner Form schon mit Javascript in Berührung gekommen und somit zu einen gewissen Grad damit bereits vertraut.

2.2.4 JackBe

JachBe Presto erlaubt die Erstellung von Mashups über einen grafischen Editor, der das Wiring Konzept nutzt, oder textuell über die XML basierte deklarative Sprache EMML. Für die Erstellung des Widgets, bei JackBe auch Mashlet genannt, bietet JackBe Presto wie WidgetBox, einige Templates über Assistenten an. Es gibt jedoch keine Möglichkeit neue Templates als Anwender zu erstellen.

2.2.5 Google Gadgets

Google bezeichnet die Widgets als Gadgets die durch die Kombination von XML und Javascript beschrieben und implementiert werden. Google stellt eine Spezifikation zur Beschreibung der Gadgets bereit, die den datenverarbeitenden Aspekt (Mashup) als Schwerpunkt und die visuelle Repräsentation (Widget) beinhaltet. Der Widget Aspekt wird in Javascript realisiert.

Kapitel 3

Auswertung

3.1 Natürliche Programmierung

Natural Programming entfällt aufgrund der Tatsache, dass es keine vollwertigen Entwicklerwerkzeuge[9] zur Optimierung gibt.

$WSO2_2$ ist interessant, da es Open-Source ist und bei der Programmierung der Mashups mittels Javascript einen hohen Freiheitsgrad erlaubt, aber damit auch eine stark eingeschränkte Zielgruppe anvisiert. Der wichtigste Aspekt, die Konstruktion von Widgets, muss vollständig manuell entwickelt werden. Es existiert keine nennenswerte Unterstützung bei der Entwicklung. Aufgrund der fehlenden brauchbaren wiederverwendbaren Elemente, muss $WSO2_2$ als Option zur Optimierung ausscheiden.

Intel Mash Maker ist der einzige, der Ansatzweise die einige notwendigen Funktionen aufweist, jedoch ist die Anwendung nicht Quelloffen und als Browser-Plugin konzipiert. Alle anderen Werkzeuge sind ebenfalls nicht Quelloffen, wodurch eine Anpassung nahezu unmöglich wird.

Bisher existieren keine Werkzeuge, die man für die Entwicklung von Widgets nutzen und natürlicher gestalten könnte. Die Option der Entwicklung einer Komponente für $WSO2_2$ resultiert in der Erstellung einer kompletten Anwendung. Eine enge Bindung an $WSO2_2$ bietet in der Hinsicht keine Vorteile.

3.2 Programming by Example

Programming by Example grenzt die Anwendbarkeit auf wiederholbare Aufgaben ein, welches in Mashup Anwendungen nicht geben ist.

[9] Entweder arbeiten sie mit Templates, welches zur einer eingeschränkten Nutzung führt, oder aber die Entwicklung artet in eine aufwendige Programmierung aus, ohne nennenswerte systemseitige Unterstützung

3.3 Domänenspezifische Sprachen

Domänenspezifische Sprache ist ein interessantes Konzept, da es erlaubt, sofern die Sprache erstmal erlernt ist, schnell Lösungen zu entwickeln. Um DSL abzuwägen, müssen zwei Fragen geklärt werden:

- Wie wird die Anwendungsdomäne definiert ?
- Wie hoch ist die Komplexität der Sprache und welche unterstützende Werkzeuge existieren bereits ?

Die Anwendungsdomäne ist die Softwareentwicklung durch Endbenutzer (EUD) im Kontext der Widgets-Entwicklung, welche auch als Rapid Web Application Development bezeichnet werden kann. Endbenutzer greifen auf fertige Bausteine, den Widgets, zurück, um schnelll und einfach neue Anwendungen zu kreieren. Die Bausteine werden dabei als Datenlieferanten und -konsumenten betrachtet, wobei der Endbenutzer nur die Steuerung des Datenflusses bestimmen muss um ein gewünschtes Verhalten zu realisieren.

Es wird eine Sprache benötigt, die den Datenstrom kontrollieren kann. Für die Kontrolle ist ein Adressierungsmechanismus mit Constraints erforderlich. Betrachten wir die Daten im Format XML, so kann XPath zur Adressierung verwendet werden, welches auch Constraints erlaubt. Die Komplexität ist somit vergleichbar mit der von XPath. XML und XPath sind bewährte Technologien, für die es bereits unzählige Werkzeuge existieren, auf die man zurückgreifen kann. DSL ist somit ein vielversprechender Ansatz.

3.4 Visuelle Programmierung

Das Herausstellungsmerkmal in der visuellen Programmierung, ist die Programmierung mit Bausteinen, welches exakt der Vorgehensweise bei der Verknüpfung von Widgets entspricht. Das Widget passt sehr gut als Darstellungsstruktur bzw. Metapher aus der Anwendungsdomäne, denn es hat einen hohen Wiedererkennungswert aufgrund der vertrauten Benutzung als Einzelanwendung.

Die Nutzung des Konzepts der visuellen Programmierung bietet sich somit für die Realisierung an. Da aber keine Widget Programmiersprache existiert, die man visualisieren könnte, muss als erstes eine entwickelt werden.

Der Aufwand der Entwicklung einer DSL und der Visualisierung ist sehr hoch, weshalb der Fokus auf DSL liegen wird.

3.5 Fazit

Die EUD-Konzepte „Natürliche Programmierung“ und „Programming by Example“ sind für das Ziel der Vernetzung von Widgets unbrauchbar. Die Mashup-Entwicklerwerkzeuge haben alle ihren Schwerpunkt auf der Entwicklung der Programmlogik, wodurch die Nutzung und Erstellung von Widgets willkürlich erscheint. Die Widgets besitzen dadurch kaum öffentliche Schnittstellen und wenn doch, dann sind diese nicht standardisiert.

Es existiert keine Sprache, die eine kontrollierte Interaktion zwischen Widgets beschreibt, weshalb die Verwendung des Konzepts der DSL hier aufgegriffen werden kann.

Das Konzept „Visuelle Programmierung“ benötigt eine Sprache die visualisiert werden soll und muss deshalb als zweiter Schritt betrachtet werden.

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

[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

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

[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

[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

[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

[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

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

[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

Abbildungsverzeichnis