Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Projektbericht

Christian Stachow

Programming for Non-Programmers

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Christian Stachow

Programming for Non-Programmers

Projektbericht eingereicht im Rahmen der Veranstaltung Projekt
im Studiengang Master Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Kai v. Luck
Zweitgutachter: Prof. Dr. Gunter Klemke

Abgegeben am 29. August 2009

Christian Stachow

Thema der Projektbericht

Programming for Non-Programmers

Stichworte

End User Development, End User Programming, Visuelle Programmierung, Natürliche Programmierung, Human Centered Interaction, Domänenspezifische Sprachen, Programming by Example, Programming by Demonstration

Kurzzusammenfassung

Aufstellung einer Übersicht über das Thema "'Programming for Non-Programmers"' und den dazugehörigen Forschungsprojekten. Vorstellung von Konzeptionsentwürfen eines universellen grafischen Editors und dessen Hürden mit einem experimentellen Prototyp.

Title of the paper

Programming for Non-Programmers

Keywords

End User Development, End User Programming, Visual Programming, Natural Programming, Human Centered Interaction, Domain Specific Language, Programming by Example, Programming by Demonstration

Abstract

Compilation of an overview of the topic "'Programming for Non-Programmers"' and their corresponding project researches. Presenting some design drafts for an universal graphical editor and its problems with an experimental prototype.

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

1.1 Werkzeuge

Ein Taschenrechner ist zweifellos ein hilfreiches Werkzeug für die Berechnung von Zahlen. Stift und Papier ergänzen es wunderbar, um Zwischenwerte festzuhalten. Ein geübter Mensch könnte dies alles im Kopf berechnen, aber je umfangreicher das Problem wird, desto schwieriger wird das Kopfrechnen. Auch ist nicht jeder Mensch gleich gut dazu in der Lage. Die Vorteile von Werkzeugen und Methoden sind die breitere Masse an potentiellen an Personen die Probleme lösen, ein effizienteres Vorgehen und die Möglichkeit neue Herausforderungen anzugehen. Selbst in der frühen Geschichte gab es Werkzeuge wie zum Beispiel der Abakus.

Seit der Einführung des Computers, eröffnet sich ein riesiger Bereich an Problemen für die passende Werkzeuge benötigt werden. Wenn man sich diesen Bereich als eine Gerade vorstellt, so stünde am Anfang der Taschenrechner für die einfache Mathematik, den selbst Kinder in der Grundschule benutzen können. Im Vergleich dazu, stünde am Ende die Software-Entwicklung für die Informatik. Software-Entwickler benötigen für ihre Arbeit Werkzeuge für die Dokumentation, die Kommunikation, die Code-Generierung, das Testen und die Organisation. Entsprechende Werkzeuge gibt bereits, jedoch in unterschiedlicher Komplexität und Speziallisierung. Ein Werkzeug für alle Variationen existiert nicht.

1.2 Ziel

Es gibt viele Begriffe wie „End User Shaping Effective Software“ (EUSES), „Programming for Non-Programmers“, „End-User Development“ (EUD), „Natural Programming“ hinter denen ein Forschungsprojekt steht. Die Forschungsprojekte haben mehr oder weniger alle das gleiche Ziel, weshalb auch die dahinter stehenden Begriffe fast schon als Synonyme betrachtet werden können.

„Programming for Non-Programmers“ oder auf deutsch „Programmieren für nicht Programmierer“ ist ein komplexes Thema. Diese Ausarbeitung soll einen Überblick über dieses Thema und den Forschungsschwerpunkten geben. Außerdem werden erste Konzeptionsentwürfe eines universellen grafischen Editors und dessen Hürden mit einem abschließenden experimentellen Prototyp vorgestellt.

Kapitel 2

Programming for Non-Programmers

„Programming for Non-Programmers“ beschreibt den Bereich der Software-Entwicklung für Personen (End-User), deren Hauptaufgabe nicht die Software-Entwicklung 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 Software-Entwicklern zugute.

2.1 End User Development

Human-Computer Interaction (HCI) ist ein interdisziplinär Gebiet[1] aus den Bereichen:

- Informatik - Anwendungsdesign, Entwicklung Benutzerschnittstellen, ...
- Psychologie - Die Anwendung von Theorien kognitiver Prozesse und empirischer Analyse von Benutzerverhalten
- Soziologie and Anthropologie - Interaktionen zwischen Technologie, Arbeit und Organisation
- „Industrial design“ - Interaktive Produkte

End-User Development[2] (EUD) ist die Entsprechung der Informatik der HCI, welcher die gängige Bezeichnung in Europa für „Programming for Non-Programmers“ ist. Lieberman u.a. (2006) definieren EUD als eine Menge von Methoden, Techniken und Werkzeugen, die Benutzern erlauben Software-Artefakte zu erstellen, zu verändern oder zu erweitern und deren Hauptaufgabe nicht die Software-Entwicklung ist.

[1] Im Grunde genommen der Oberbegriff zu EUD, jedoch gibt es keine einheitliche Definition was HCI ist

[2] Auch als End User Programming (EUP) bekannt

2.2 Forschungsprojekte

Für die Erforschung des Themas „Programming for Non Programmers“ existieren vier größere Forschungsprojekte mit im wesentlichen überlappenden Zielen: Dem End User zu ermöglichen hochqualitative und sichere Anwendungen zu erstellen.

2.2.1 EUSES - End User Shaping Effective Software

„End User Shaping Effective Software“ (EUSES) ist hauptsächlich eine amerikanische Forschung in Zusammenarbeit mit verschiedenen Universitäten. Sie versuchen die Methodologien der Software-Entwicklung zu vereinfachen, so das selbst Laien qualitativ hochwertige Lösungen entwickeln können. Die Internet-Seite ermöglicht einen strukturierten Zugriff auf bestehende englisch sprachliche Arbeiten, Forschungen und Prototypen.

EUSES betrachtet „End User“ als Personen mit keiner bis zu einer formalen breiten Ausbildung in der Programmierung. Darunter fallen Grafik-Designer, Besitzer von Familienbetrieben, Ärzte etc. Das generelle kennzeichnende Merkmal ist die Erzeugung, Wartung und Debugging von Werkzeugen die das jeweilige Ziel unterstützen.

2.2.2 EUD Net - Network of Excellence on End User Development

Das „Network of Excellence on End User Development“ (EUD Net), ist das europäische Gegenstück zu EUSES. Die Internetpräsenz von EUD Net bot öffentlich Publikation aus Konferenzen, Zeitschriften, etc. zur freien Verfügung an. Leider ist zum Zeitpunkt des Verfassens dieser Ausarbeitung die Internetpräsenz nicht mehr verfügbar. Der Grund für das Verschwinden ist unbekannt.

2.2.3 EUDISMES - End User Development in Small and Medium Enterprise Software Systems

Ein von der Universität Siegen geführtes deutsches Projekt namens „End User Development in Small and Medium Enterprise Software Systems“ (EUDISMES), befasst sich mit der Entwicklung von innovativen Techniken des End User Development's für kleine und mittelständische Unternehmen. Dieses Projekt hat eine Laufzeit von gut drei Jahren die von Oktober 2005 bis September 2008 lief und vom Bundesministerium für Bildung und Forschung (BMBF) gefördert worden ist. Der besondere Praxisbezug dieses Projekts wird durch die Beteiligung neben der SAP AG und Buhl Data als auch durch kleine und mittelständige Anwenderunternehmen deutlich. Ergebnisse der Arbeiten sind auf deren Internetseite veröffentlicht.

2.2.4 Natural Programming Project

Das „Natural Programming Project“ (NatProg; Myers, 1998) arbeitet daran, die Programmiersprachen und Entwicklungsumgebungen leichter erlernbar, effektiver und sicherer zu machen. Dabei wird der Fokus auf den Menschen gesetzt. Es findet keine Unterscheidung zwischen unerfahrene, Anfänger oder professionelle Programmierer statt. Aktuelle Prototypen reichen von mit Meta-Informationen angereicherten Java API-Dokumentationen, Programmfluss-Analysen bis hin zu innovativen Debugging-Alternativen. Zu den einzelnen Projekten werden weiterführende Literaturen aufgeführt und zum Großteil auch als Download angeboten.

2.3 Anforderungen

Sutcliffe u. a. (2003) betrachten End-User Development als Outsourcing der Software-Entwicklung und beschreiben die wesentlichen 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. Folgende wichtige Eigenschaften muss so ein Werkzeug erfüllen:

- Geringer Lernumfang - Ein neues Werkzeug erfordert einen Initialaufwand für die Einarbeitung, die vom End-User als Hürde wahrgenommen wird. Die Effizienzsteigerung muss wahrnehmbar sein.
- Wiederverwertbarkeit - Ist das Werkzeug nur für seltene Problem einsetzbar, so stellt sich die Frage nach der „globalen“ Effizienz.
- Fehleranfälligkeit - Die Folgen von Fehler bei der Nutzung oder beim Lernen beeinflussen den End-User stark.

Weiterhin stellen sie folgende Grundsätze auf:

1. Die Benutzerzufriedenheit eines allgemeinen Programms ist umgekehrt proportional zu der Komplexität des Produkts und der Variabilität der Benutzergruppen.
2. Anstrengungen der Benutzer von Anpassungen und das Erlernen der Software ist proportional zu der wahrgenommenen Nützlichkeit eines Produkts in Erfüllung eines Arbeitsauftrags oder der Unterhaltung.
3. Die Akzeptanz der Adaption ist umgekehrt proportional zu Systemfehlern in der Adaption mit der Folge, dass unangemessene Adaptionen unsere Motivation negativ beeinflussen (Kosten der Diagnostizierung von Fehlern, Kosten von Umwegen und negative Gefühle).

2.4 Kategorien

Bei der Forschung über Methoden, Techniken und Werkzeuge haben sich im wesentlichen vier Kategorien etabliert.

2.4.1 Visual Programming

Bei der Visuellen Programmierung kann der Benutzer durch das Zusammensetzen von vorgegebenen „Programmier-Bausteinen“ sein eigenes Programm programmieren. Der Benutzer muss keine Syntax lernen. Er programmiert durch das Zusammensetzen von Bausteinen.

Martin Sukale (Sukale, 2008) stellt mehrere Entwicklungsumgebungen (LabVIEW, PureData, . . .) im Kontext der multimedialen Datenverarbeitung vor. Green und Petre (1996) analysieren die Nutzbarkeit von „Visual Programming“ Entwicklungsumgebungen. 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.

Seine zusammenfassende Schlussfolgerung kann ich jedoch nicht teilen:

> Zur Erstellung komplexer Programmstrukturen ist VP wegen der schlechten Skalierbarkeit nur bedingt geeignet. Motivierende Interaktionsmechanismen (z.B. direkte Manipulation und ansprechende Symbolbilder) können zur Belastung werden, wenn große Komponenten zu erstellen sind. Petre [19] zitiert einen ernüchterten Programmierer mit folgenden Worten: „I quite often spend an hour or two just moving boxes and wires around, with no change in functionality, to make it that much more comprehensible when I come back to it.“ Auf der Suche nach Ansätzen zur professionellen und rationellen Entwicklung allgemeiner Softwaresysteme ist VP deshalb als Sackgasse zu beurteilen.

Meine Stellungnahme zu seinen Punkten:

1. Der erste Punkt ist verständlich, da in der textuellen Programmierung Techniken wie Polymorphie, Komponentenbildung etc. existieren und entsprechende Gegenstücke in der visuellen Programmierung nur bedingt vorhanden sind.

2. Der zweite Punkt kommt so auch in der textuellen Programmierung vor. Software-Entwickler ändern den Code um diesen lesbarer zu gestalten (Strukturverbessernde Wartung, Kommentare,. . .).

3. Sackgassen gibt es nicht, sondern die Frage ist, wie gut die Strassen gepflastert sind bzw. werden.

Marten[3], LabVIEW, PureData, Croquet Croquet und Alice Alice sind beispielhafte Umsetzungen der Programmier-Metapher der visuellen Programmierung.

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 Abb. 2.1) 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.

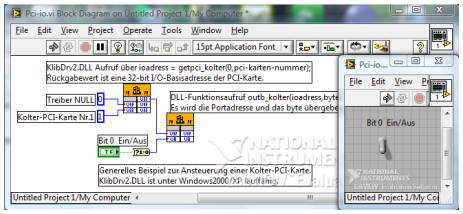

Abbildung 2.1: LabVIEW Datenfluss und Bedienelemente

Das Croquet SDK ist in eine von Smalltalk abgeleiteten Programmiersprache namens Squeak verfasste 3D Multibenutzer-Entwicklungsumgebung. Mit ihr lassen sich virtuelle gemeinschaftliche Echtzeit-Welten entwickeln und verteilen. Programmiert wird textuell wie in altbekannten Entwicklungsumgeben, jedoch kann man in einer drei dimensionalen Welt mit sofortigem Feedback (siehe Abb. 2.2) gemäß „WYSIWYG - What You See Is What You Get“ (Was du siehst, ist was du bekommst) programmieren.

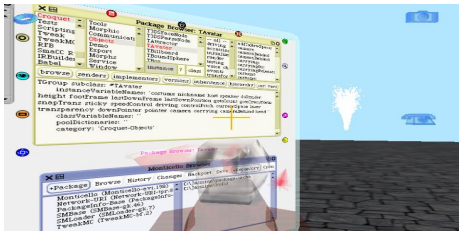

Abbildung 2.2: Croquet Quelltext Editierung

Alice (siehe Abb. 2.3) ist eine objektorientierte Programmiersprache, welche erlaubt Objekte und Funktionen durch Drag&Drop zu nutzen, um 3D Animationen zu erstellen. Das Ziel von Alice ist Unerfahrenen die Grundkonzepte der Programmierung im Kontext von animierten Filmen und einfachen Videospielen zu lehren. Dabei programmiert der Unerfahrene das Verhalten von 3D Objekten (z.B. Menschen, Tiere und Autos) in einer bevölkerten virtuellen Welt.

2.4.2 Natural Programming

Natural Programming (Myers u. a., 2004) versuchen Programmiersprachen und Entwicklungsumgebungen natürlicher bzw. näher an die Betrachtungsweise von Menschen über Aufgaben zu gestalten. Ihr Ziel ist die Möglichkeit zu schaffen, dass die Menschen ihre Ideen so ausdrücken können wie sie darüber nachdenken. Eine Art die Programmierung

[3]MacOSX Version einer visuellen Software-Entwicklungsumgebung: http://www.andescotia.com/products/marten/

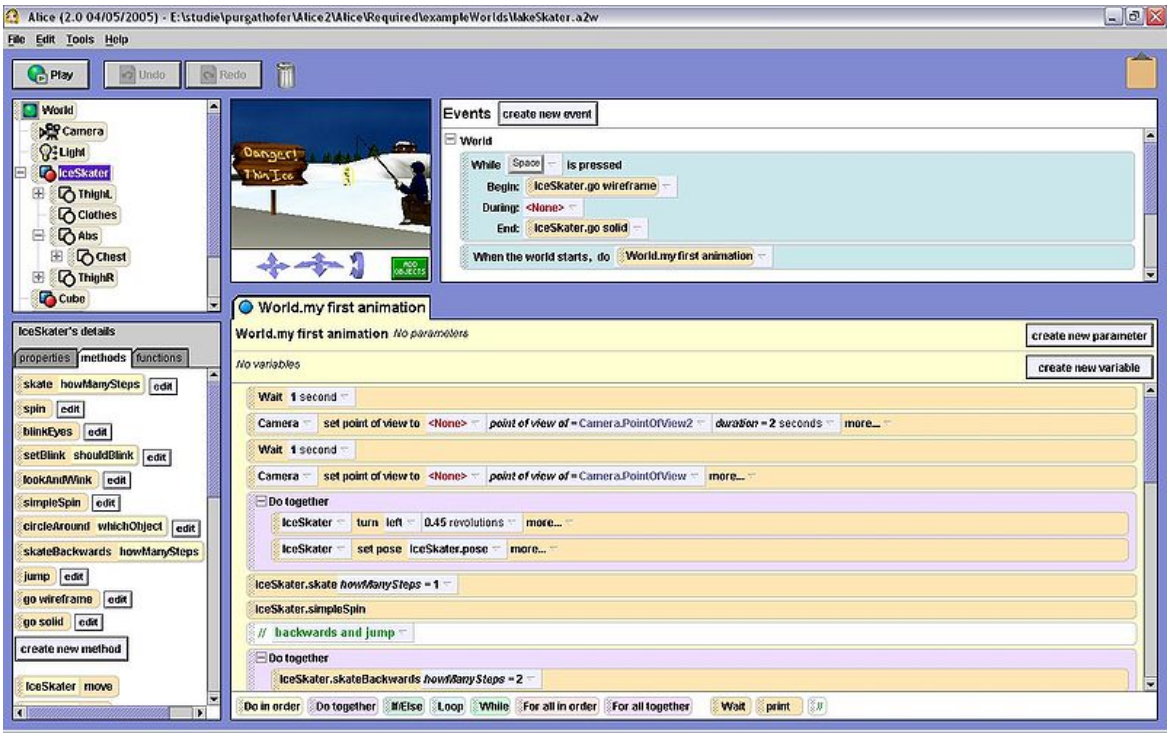

Abbildung 2.3: Alice Entwicklungsumgebung

zu definieren lautet: Der Prozess der Transformation eines mentalen Plans aus bekannten Begriffen in eines für den Computer kompatiblen Form.

Eine Innovation betrifft das Debuggen von Programmen. Bei falschen Programmverhalten fragt man sich häufig *warum dies* oder *warum dies nicht* geschieht. Typische Entwicklungsumgebungen erlauben diese Art von Fragestellung nicht. Alice (siehe Abb. 2.4) erlaubt dies. Dabei wird der mit Meta-Informationen angereicherte Programmfluss grafisch dargestellt.

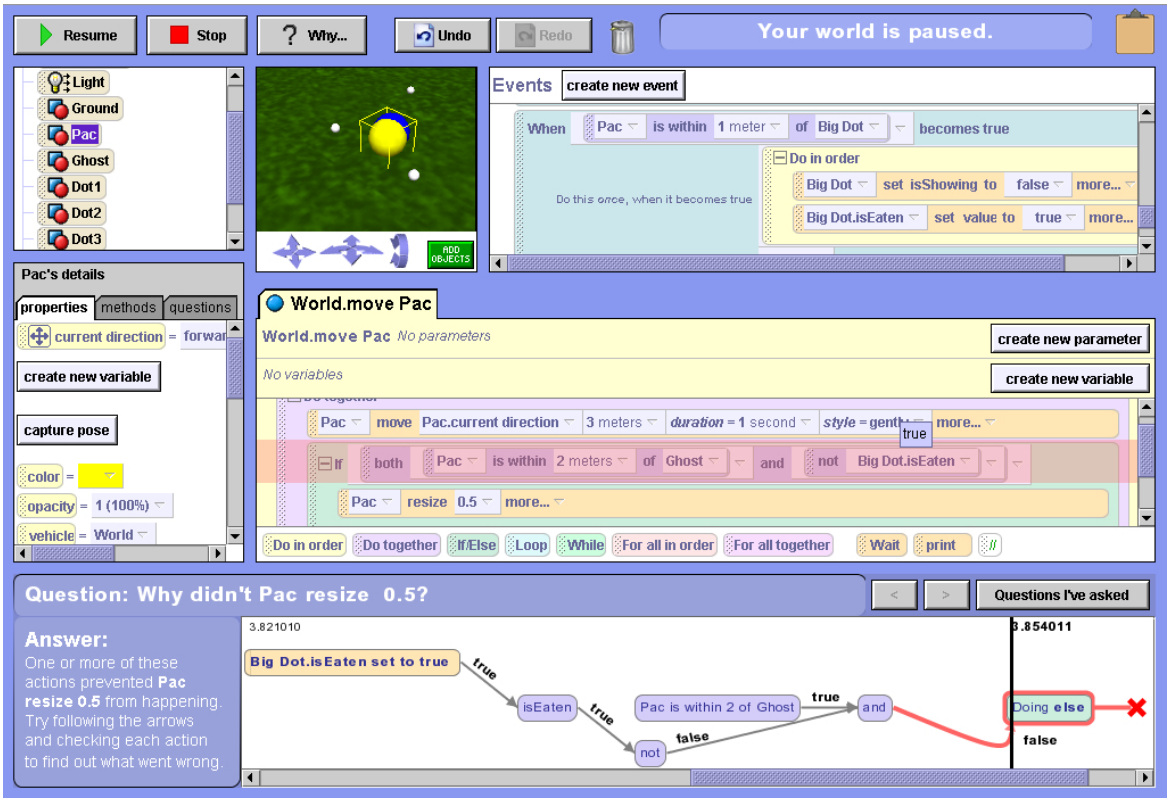

Abbildung 2.4: Alice WhyLine

2.4.3 Programming by Demonstration/Example

Der Begriff „Programming by Demonstration“ bzw. „Programming by Example“ (Lieberman, 2001; Cypher, 1993) 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 was der Benutzer macht, während er normale Befehle erteilt. Dadurch programmiert der Benutzer durch ein Beispiel 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 fehlt 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 „Macro“, in denen 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 Texteditor UltraEdit. Er erlaubt Befehle aufzuzeichnen die ausgeführt werden und später beliebig oft genutzt werden können.

Cypher (1993) grenzt 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.

2.4.4 Skript- und domänenspezifische Sprachen

Skriptsprachen (Morin und Brown, 1999) sind als solche schwierig zu definieren, da sie sich mit der Zeit weiterentwickeln. Angefangen mit JCL[4] bis hin zu Perl, die in ihrer Mächtigkeit den „third generation languages“ (3GL) 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.

Domänenspezifische Sprachen (DSL) charakterisiert Paul Hudak in „Modular Domain Sepcific Languages and Tools“ (Hudak, 1998) als eine Programmiersprache die maßgeschneidert für eine besondere Anwendungsdomäne ist. Charakteristika von einer optimalen

[4] 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).

DSL ist die Fähigkeit, ein vollständiges Anwendungsprogramm für eine Domäne schnell und effizient zu entwickeln. Ein 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.

Kapitel 3

Piktor - Piktogramm Editor

3.1 Idee

Die Idee hinter dem „Piktor“ ist die Entwicklung eines universellen graphischen Editors. Einen „Texteditor“, der Piktogramme verwendet. Es existieren schon Produkte, die der Vision nahekommen. Zum Beispiel „LabVIEW“ von National Instruments oder MathModelica von MathCore. Leider sind diese Produkte kostenpflichtig und auf bestimmte Anwendungsgebiete zugeschnitten und somit nur bedingt universell verwertbar.

Die Open-Source Lösung Open Knowledge Simulation Modeling (OKSIMO) erlaubt beliebige Prozesse beliebiger Anwendungsdomänen zu modellieren und diese zu simulieren. Da ich erst gegen Ende dieser Ausarbeitung auf dieses vielversprechende Projekt gestoßen bin, bleibt der mögliche Wechsel auf OKSIMO als Plattform nur als Ausblick offen.

Weil kein anderes Produkt frei zugänglich oder „universell“ nutzbar war, entschloss ich mich deshalb zur der Entwicklung einer eigenen Plattform bzw. Editors.

3.2 Konzeption

Das Konzept steht noch nicht vollständig fest, da ein reduzierter Prototyp über die Machbarkeit und Komplexität Aufschluss geben soll und einige Ideen gegenwärtig visionär anmuten. Deshalb realisiert der Prototyp nur einige Punkte des Konzepts.

Die Piktor-Plattform besteht aus 3 Komponenten die im folgenden erläutert werden.

3.2.1 Editor / Modellierung

Ein Modell besteht nur aus zwei Objekttypen:

- **Funktion / Funktionsblock** - Eine Funktion kann beliebig viele Ein- und Ausgänge

besitzen. Der Ausgangswert und -typ wird in Abhängigkeit der Logik der Funktion und der Eingänge ermittelt.

- **Verbindung** - Verbindungen vernetzen Funktionsblöcke miteinander, worüber Daten fließen. Daten fließen nur von einem Ausgang zu einem Eingang, die vom gleichen Typ sind.

Der Editor wird eine Bibliothek an vordefinierten Funktionen besitzen, aus denen eigene Algorithmen erstellt werden können. Selbst erstellte Algorithmen können als eine Art Blackbox extrahiert werden und als eigenständige Funktion zur Verfügung gestellt werden. Der ausgewählte Bereich wird dabei in einen Blackbox-Editor (siehe 3.1) übertragen und von einem Rechteck eingegrenzt. Man platziert die benötigten Ein- und Ausgänge und verdrahtet diese je nach Wunsch. Auf diese Weise können häufig verwendete Algorithmen bzw. Funktionen leicht wiederverwendet werden und die Lesbarkeit wird stark verbessert. Wünschenswert wäre ein Blackbox-Automatismus ähnlich wie dem Code-Folding aus Quelltext-Editoren, um die Übersicht zu verbesseren.

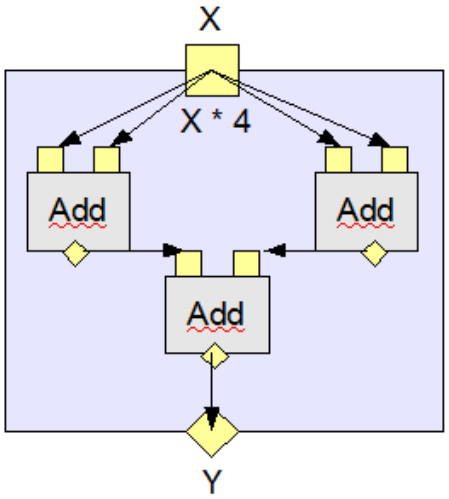

Abbildung 3.1: Blackbox Funktionsentwurf

Der Editor muss zwingend über folgende Werkzeuge verfügen:

- Funktion / Funktionsblock erstellen
- Propertyeditor - Eigenschaften von Objekten (Funktionen und Verbindungen) ändern
- Verbindung erstellen
- Objekt löschen
- Objekt verschieben
- Perspektive verschieben

Ohne diese rudimentären Werkzeuge wäre ein Erstellen eines Modells unmöglich.

Folgende Werkzeuge erleichtern das Arbeiten mit dem Editor und erhöhen die Produktivität des Benutzers (ganz im Sinne des Programming for Non-Programmers Paradigmas):

- Zoom - Je nach Positionierung des Cursors, wird an der Stelle zentriert gezoomt. Dies erlaubt auch alternativ die Perspektive zu verschieben.

- Minimap - Sobald ein Modell Größen annimmt, die nicht mehr überschaubar sind, erleichtert eine miniaturisierte Ansicht des Modells die Orientierung und Positionierung.
- Blackboxing - Eine Gruppe von verbundenen Funktionen kann als eigenständige Funktion extrahiert werden, welches eine Wiederverwendung erlaubt und im besonderen die Übersicht verbessert.
- Auto-Blackboxing - Je nach Zoom-Faktor oder frei wählbar, wird ein automatisierter Blackboxing-Algorithmus über Gruppen von Funktionen mit bestimmten Voraussetzungen verwendet. Mögliche Voraussetzungen könnten sein:
  - Dichte bzw. Nähe von Funktionen
  - Falls das Zoomen die sichtbare Größe der Funktionen nicht beeinflusst, dann können Funktionen aneinander stoßen → Blackboxing
  - Markierungen vom Benutzer vorgegeben
- Task-Focusing - Soll Informationsüberladungen vermeiden. Für eine Aufgabe unrelevante Funktionen und Bereiche werden ausgeblendet. Mylyn ist eine solche Umsetzung für Eclipse und dort können Konzepte und Ideen übernommen werden.
- Gesten Steuerung - Undo / Redo könnten über das Bewegen der Maus nach links respektive rechts ausgeführt werden. Das Löschen selektierter Objekte könnte über das Bewegen der Maus nach unten geschehen. Eine Geste für die Erzeugung von Objekten bietet Spielraum für viele Ideen die in 3.2.4.1 genauer behandelt wird.

Um automatisiert Daten zu verarbeiten und die Anwendungsisolierung[5] zu vermeiden, wird ein Zugriff auf externe Quellen wie aus Datenbanken, Dateisystemen oder anderen Anwendungen (z.B. Excel) benötigt.

3.2.2 Simulator

Nach der Erstellung eines Modells, soll dieses schließlich auch simuliert werden. Es stellt sich erstmal die Frage nach der Art der Simulation und deren Anforderung.

- Wird ein Modell sequentiell oder parallel verarbeitet ?
- Wie scharf sind die Echtzeitanforderung, wenn überhaupt gegeben ?
- Anstatt eine eigene Virtuelle Maschine zu entwickeln, kann auf eine bestehende zurück gegriffen werden ?

[5] Die Vernetzung mit anderen Anwendungen ist von essentieller Wichtigkeit, denn die wenigsten würden sich an Insellösungen binden

- Wie wird ein Modell ausgeführt ?

Die Fragen, bis auf die Art der Ausführung, bleiben in dieser Ausarbeitung unbeantwortet. Der Grund dafür sind die vielen Folgeprobleme, die im Zusammenhang mit der End-User Schnittstelle stehen. Glücklicherweise findet sich dort auch ein Schlupfloch, der den Simulator in bestimmten Situationen überflüssig macht. Genaueres hierzu findet sich im nächsten Abschnitt 3.2.3.

Der Simulator erlaubt das ausführen eines Modells, ähnlich einer virtuellen Maschine. Die Idee beruht auf die Tatsache, dass der Editor ein „Code-Modell“ erzeugt, welches in eine ausführbare Form übersetzt und ausgeführt werden kann. Ein konkretes Beispiel wäre die Robotik[6]. Roboter besitzen Sensoren (Input) und Aktoren (Output) die irgendwie miteinander in Beziehung stehen. Diese Beziehung könnte mit dem Piktor beschrieben werden, jedoch kann der Roboter das Piktor-Modell nicht verstehen. Das Piktor-Modell muss in eine für den Roboter taugliche Form transformiert werden. Folgende drei Möglichkeiten kämen in Betracht:

- Nativer Maschinencode - Das Piktor-Modell wird für den Roboter verständlichen nativen Maschinencode transformiert und überspielt.
- Virtueller Bytecode - Verhält sich wie nativer Maschinencode, jedoch ist das Ziel eine virtuelle Maschine die auf dem Roboter läuft.
- Remote Control - Die Berechnung laufen auf einem Server, welcher über wohl definierte Kommandos den Roboter steuert. Der Roboter schickt dabei periodisch seine Sensorwerte an den Server.

3.2.3 Preview / End-User Schnittstelle

Der Editor beschreibt nur Beziehungen zwischen Informationen und deren Verarbeitung. Um die Informationen für den End-User nutzbar zu machen, wird eine End-User Schnittstelle benötigt. Falls keine bidirektionale Kommunikation mit externen Schnittstellen (z.B. Excel) möglich ist, so muss eine Domänen gerechte Schnittstelle zur Verfügung gestellt werden. Der End-User muss in der Lage sein, Auswirkungen seiner Änderungen seines Modells sofort wahrzunehmen (WYSIWYG), ohne das es zu Informationsüberlastungen kommt. Eine Änderung eines Wertes darf nicht zu tausend blinkenden Positions verändernden Elementen führen und so für Verwirrung sorgen. Das Problem der Erzeugung einer End-Users gerechten und anwendungsspezifischen Schnittstelle ist ein Grundproblem des „End-User Developments“, für die es bis heute noch keine zufriedenstellende Lösung gibt. Eine treffende Metapher für das Dilemma ist:

[6] Lego Mindstorm NXT ist ein kommerzielles Beispiel

Da beißt sich der Hund in den Schwanz.

Die Alternative und zugleich ein Schlupfloch aus dem Dilemma, ist die Integration bestehender Anwendungen. Der Vorteil der Integration ist naheliegend: Vertrautes Umfeld des End-Users. Der Aufwand der Entwicklung der Piktor-Plattform würde sich dadurch drastisch reduzieren. Die Simulator-Komponente würde entfallen und die End-User Schnittstelle würde sich auf spezielle Adapter für Anwendungen reduzieren. Der Piktor-Editor hätte damit einzig die Funktion einer alternativen Datenrepräsentation und -manipulation, denn der Piktor-Simulator befände sich in der externen Anwendung.

Ziehen wir als konkretes Beispiel die Tabellenkalkulation Excel heran. Der End-User erstellt in Excel seine Tabellen und versieht sie mit Formeln. Tage später sollen die Tabellen erweitert werden, und der Zusammenhang der Formeln ist, aufgrund des Kontextwechsels, nicht mehr offensichtlich. Der Piktor-Editor bietet alle relevanten und verdeckten Information an und erlaubt dem End-User so sich schnell dem Kontext anzupassen. Die Tabellenlogik wird entsprechend angepasst und anschließend mit Excel synchronisiert. In Excel können dann Prognosen, Berichte und dergleichen evaluiert werden.

3.2.4 Offene Punkte

3.2.4.1 Objekterzeugung durch Gesten

Die Suche nach Objekten über Beschreibungen sind langatmig und Identifikationsbezeichner ein Relikt aus der textuellen Programmierung. Eine angemessene Suche wäre über die Bestimmung der grafische Äquivalenz eines gezeichneten Objekts oder durch Gesten (Buxton, 2009). Dabei geh ich von folgenden Annahmen aus:

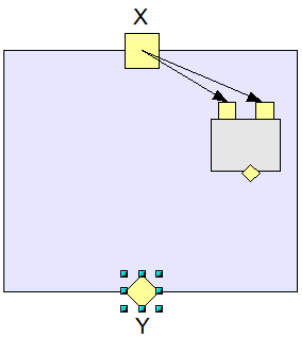

Abbildung 3.2: Skizze einer gesuchten Funktion

- In der grafischen Entwicklung sind Piktogramme die häufigsten genutzten Elemente und somit am stärksten in der Erinnerung des Menschen verankert.
- Objekte, in diesem Fall Funktionen, besitzen eine eingeschränkte Auswahl an Darstellungsmöglichkeiten (Kurven und Geraden)
- Funktionen (Piktogramme) sind charaktersierbar.
  - Wie viele Geraden im welchen Verhältnis zueinander an welchen Stellen ?
  - Wie viele Kurven mit welcher Steigung an welchen Stellen ?
  - Wie viele Ein- und Ausgänge und wo platziert ?

  - Welche geschachtelten Funktionen und wo platziert ?
  - Welcher Datentyp wird verarbeitet ?

- Benutzerfreundliche Fehlertoleranz bei der Erkennung von Piktogrammen. Eine gerade Linie eines Betrunkenen soll als gerade Linie erkannt werden.

Während sukzessive das gesuchte Piktogramm (Funktion) Freihand gezeichnet wird, erscheint eine sortierte Auswahl an Treffern. Passende Bereiche werden dabei in Grün, falsche in Rot und nicht zugeordnete in Schwarz dargestellt. Die Abweichungen werden in unterschiedlichen Stufen dargestellt. Je stärker die Abweichung desto stärker geht der Grünton in ein Rot über. Ein Tooltip stellt den Kandidaten in Normalgröße und mit detaillierten Informationen dar.

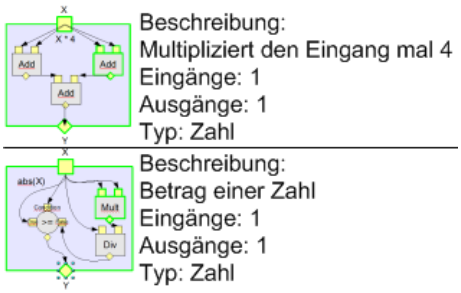

Abbildung 3.3: Trefferliste zur Skizze

3.2.4.2 Nicht näher behandelte Punkte

Offene Punkte die zwar erfasst, aber nicht weiter behandeln werden sind:

- Modelldefinition - Was bildet ein Modell genau ab ?
- Kontrollstrukturen - Schleifen, Rekursion, Bedingungen etc. Welche werden gebraucht ?
- Wiederverwendbare Eingänge - Die Addition kann theoretisch beliebig viele Eingänge haben, jedoch muss bei der Subtraktion die Reihenfolge Aufgrund der Vorzeichen beachtet werden $\rightarrow a - b$ und nicht $-a - b$

3.3 Frameworks

Im folgenden werden die für den Editor sinnvollen und teilweise verwendeten Frameworks kurz vorgestellt.

3.3.1 Eclipse Modeling Framework (EMF)

Die Verwendung mit dem Graphical Editor Framework(Moore u. a.) wird meistens in Zusammenarbeit mit dem Eclipse Modeling Framework. Jedoch wurde zu Gunsten einer

schnelleren Einarbeitung des GEF auf die Nutzung von EMF verzichtet. Wikipedia[7] beschreibt EMF folgendermaßen:

> EMF kann aus einem Modell Java-Code erzeugen. Das so erzeugte Java-Programm kann Instanzen dieses Modells erstellen, abfragen, manipulieren, serialisieren (eingebaut als XMI oder anderes XML, mit Plugin auch in einer relationalen DB), validieren und auf Änderungen überwachen (für MVC). Darüber hinaus wird JUnit-Code erzeugt, der den generierten Code testet. Das Modell selbst kann aus einer XSD (wie etwa bei JAXB), aus annotierten Java-Interfaces oder aus UML-Diagrammen (Rose, Magic Draw und Omondo) generiert werden, oder auch von Hand (mit einem „Baumeditor“) erstellt werden. Der aus dem Modell generierte Code umfasst den eigentlichen Modell-Code (wie ihn etwa JAXB erzeugt), Code für Wizards, Editoren, bis hin zum Code für die eigentliche RCP-Anwendung. Das Modell selbst, die Generierung daraus sowie der generierte Code können angepasst werden, implementierte Funktionalität und neu generierter Code werden dabei gemerged. . . Für weitergehende Ansprüche bietet EMF etwa die Möglichkeit, Modelle dynamisch zur Laufzeit zu generieren (etwa wenn erst dann das Modell bekannt ist).

3.3.2 Graphical Editor Framework (GEF)

Die Eclipse-Erweiterung Graphical Editor Framework erlaubt es anhand eines existierenden Datenmodells einen dazugehörigen grafischen Editor relativ schnell zu erstellen. Es folgt dem Model-View-Controller (MVC) Muster. Das „Model“ wird extern erstellt (z.B. über EMF) und besitzt eine Baumstruktur. Die „View“ nutzt die Draw2D API für das Zeichnen von Figuren. Eine Figur repräsentiert ein Element im Datenmodell. Die Figuren spiegeln in ihrer Struktur die des Datenmodells. Der „Controller“ und somit das Bindeglied zwischen dem Modell und den Figuren sind sogenannte „EditParts“. Für jedes Datenelement existiert ein EditPart und eine Figur. Das bedeutet, dass das Modell, alle EditParts und alle Figuren typischerweise gleichartig strukturiert sind. Ein EditPart reagiert auf Benutzer-Interaktionen und über sogenannte „Policies“. Eine Benutzer-Aktion stößt einen „Request“ an, die an passende Policies geleitet werden. Jede Policy kann dabei darauf reagieren und ein „Command“ bereitstellen, welches später ausgefürt wird. Datenmanipulationen werden ausschließlich über „Commands“ realisiert. Das ermöglicht eine einfache Umsetzung der „Undo“ und „Redo“ Funktionalität. EditParts „lauschen“ auf Veränderungen im Modell und passen bei Bedarf ihre Figuren an.

Das Buch „Eclipse Development using the Graphical Editing Framework and the Eclipse Modeling Framework“ (Moore u. a.) erklärt ausführlich die Funktionsweise von GEF mit vielen Illustrationen und Beispielen.

[7] http://de.wikipedia.org/wiki/Eclipse_Modeling_Framework

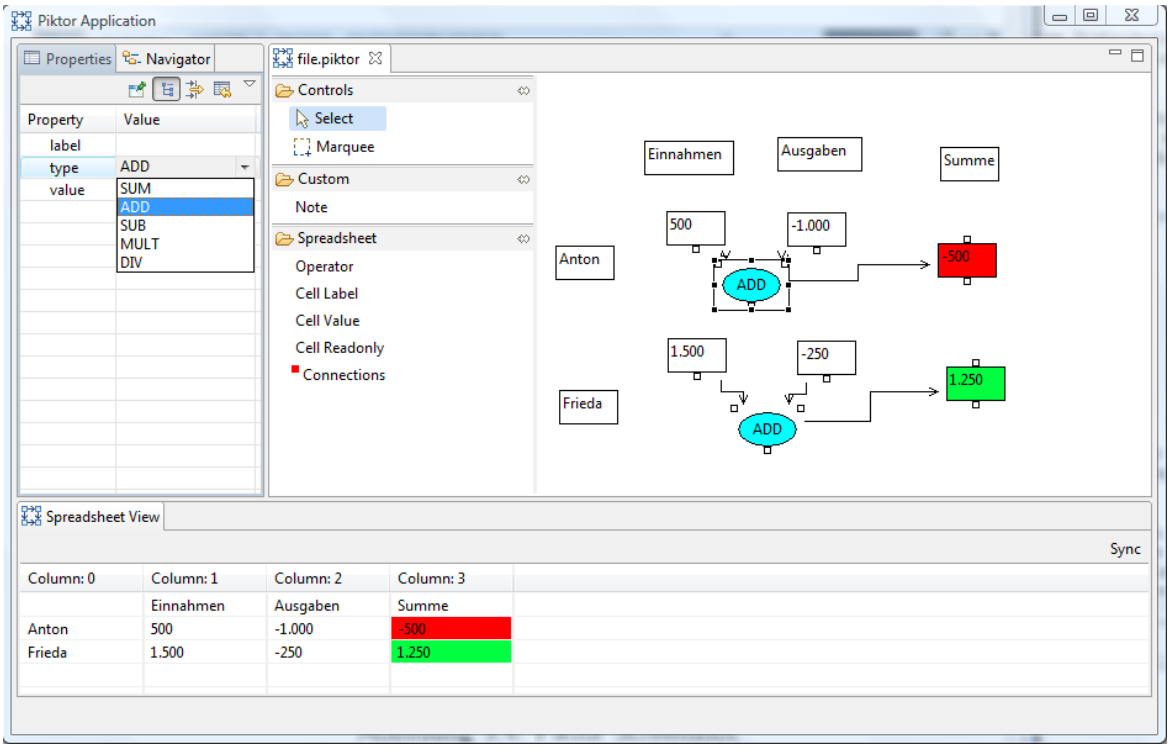

Abbildung 3.4: Piktor Screenshot

3.3.3 Eclipse Plug-in Development Environment (PDE)

Eclipse ist eine durch Plug-ins erweiterbare Plattform. Plug-ins können bestehende Eclipse Distributionen oder ein leeres Eclipse derart erweitern, dass maßgeschneiderte Rich-Client Lösungen möglich sind. Eclipse nutzt das eigenständig entwickelte OSGi-Framework[8] Equinox um Plug-Ins bzw. Bundles zu managen.

Das wichtigste bei einer Plug-in Entwicklung[9] sind die beiden Dateien „Manifest.mf“ und „plugin.xml“. Im Manifest wird die Identität des Plug-ins aufgeführt und welche Abhängigkeiten zu anderen OSGi Plug-ins bestehen. Für die Integration in Eclipse (UI,Events, etc.) müssen Extension-Points in einer „plugin.xml“ deklariert und mit entsprechenden Java-Klassen verbunden werden.

3.4 Prototyp

Der Prototyp namens Piktor (siehe Abb. 3.4) imitiert einen kleinen Teil einer Tabellenkalkulation. Als End-User Schnittstelle wird keine externe Tabellenkalkulation verwendet, sondern eine einfache, interne und editierbare Tabellenansicht.

Piktor ist als ein Eclipse Plug-In konzipiert, um auf eine einfache Art und Weise das GEF-Framework zu nutzen und eine Rich-Client Anwendung zu erstellen. Nach anfänglichen Einarbeitungsschwierigkeiten und das Einbinden von Fremd-Bibliotheken[10], machte ich recht schnelle sichtbare Fortschritte. Mit wenigen Handgriffen und einem Minimalbeispiel als Vorlage, ließ sich schnell das Grundgerüst aufbauen. Aufwendig wurde es, sobald Verhalten umgesetzt werden sollte, welches sichtlich vom Beispiel abweicht. Das stöbern im „Eclipse Development using the Graphical Editing Framework and the Eclipse Modeling Framework“ (Moore u. a.) bot meist eine zufriedenstellende Antwort.

Das GEF-Framework sieht für das Benachrichtigen bei Modelländerungen das Observer-Pattern vor. Um das Benachrichtigen zu automatisieren und so Tipp- und Fleißarbeit zu sparen, nutze ich *cglib*. Alle modellverändernde Methoden werden abgefangen und dabei alle Beobachter implizit benachrichtigt. Jedes Modell wird über eine *Factory* erstellt und besitzt einen *protected* Konstruktor, um zu garantieren, dass der Benachrichtigungsautomatismus als AOP[11] mit *cglib* implantiert ist.

Dynamisch erzeugte und bewegliche Ein- und Ausgänge (Ports genannt) verursachten ungeahnte Probleme, die bis heute nicht gelöst sind. Die aktuelle Fassung arbeitet mit statischen Ein- und Ausgängen. Folgende Möglichkeiten habe ich ausprobiert:

1. Ein- und Ausgänge sind Unterelemente einer Funktion

2. Ein- und Ausgänge sind unabhängige Elemente einer Funktion

Im ersten Fall tritt folgendes ungelöstes Problem auf. Jede Figur besitzt einen Bereich in dem es zeichnen darf. Dieser Bereich ist immer innerhalb der Eltern-Figur. Ports sollen nicht die Eltern-Figur übermalen oder unter ihnen verschwinden, sondern an ihr „andocken“. Man benötigt also eine Eltern-Figur die immer einen Rand für die Ports frei lässt. Ports sind normale Figuren und müssen irgendwie als Sonderfall[12] behandelt werden. Weiterhin

[8] http://www.osgi.org/

[9] http://wiki.eclipse.org/PDE

[10] Bibliotheken, für die es kein OSGi-Bundle gibt, bergen großes Gefahrenpotential aufgrund von überlappenden Abhängigkeiten

[11] Aspektorientierte Programmierung - Erlaubt bestehenden Code zu erweitern, ohne diesen ändern zu müssen.

[12] GEF nutzt für das Platzieren der Figuren sogenannten *Layout-Manager*. Ein spezieller *Layout-Manager* löste das Problem bedingt. Die Eltern-Figur besaß einen *XYLayout-Manager* der auch bei einigen Kindern erwartet wurde → *Class Cast Exception*. In der begrenzten Zeit habe ich die Ursache nicht festellen können.

müssen nicht alle Ports gleich aussehen. Das größte Problem ist aber das Verschieben der Ports. Aus einem mir unerklärlichen Grund lassen sich die Ports nur innerhalb der Figur (ohne Rand) bewegen. Da GEF überwiegend ereignisgesteuert ist, ist ein Debuggen äußerst schwierig.

Der zweite Fall beinhaltet das Problem der Zuordnung der Ports zu Figuren. Ein *Operater* muss seine Eingänge kennen, um diese nach Werten abzufragen. Jedoch müssen die Ports als Wurzel-Datenelement vorliegen, damit GEF diese als unabhängige Figuren behandelt. Redundante Referenzen zu managen ist ein unsauberer Programmierstil und deshalb wird diese Variante verworfen.

Einige Objekte sind in der Lage direkt innerhalb des Editors ihren Wert zu ändern. Dazu muss nur innerhalb des bereits selektierten Objekts ein Linksklick ausgeführt werden. Nach knapp einer Sekunde wird dann das Objekt durch ein mehrzeiliges Textfeld überlagert, in dem die Eingabe erfolgt. Nach Abschluss der Eingabe reicht ein neu setzen des Fokus außerhalb des Objekts für die Eingabeübernahme.

3.4.1 Implementierte Objekte

In der aktuellen Fassung existieren sechs Objekte und mit dem Property Editor editierbaren Eigenschaften:

- Note - Ein *Notizzettel* um Beschreibungen zu platzieren
  - *label* - Beschreibung (Not yet used)
  - *value* - Beschreibung die dargestellt wird (Veraltet, wird durch die Eigenschaft *label* ersetzt)
- Operator - Je nach Einstellung berechnet sie die beiden Eingänge und stellt den Wert am Ausgang bereit. Die Berechnungsvorschrift läßt sich auch über das Kontextmenü (Rechtsklick) ändern.
  - *label* - Beschreibung (Not yet used)
  - *type* - Bestimmt die Berechnungsvorschrift (Standard *Add*)
  - *value* - Aktueller berechneter *readonly* Wert der am Ausgang anliegt (Standard 0).
- Cell Value - Eine Zelle die vom Benutzer mit einer Zahl versehen werden kann. Sie Besitzt einen Ausgang.
  - *backgroundColor* - Hintergrundfarbe
  - *column* - Abgebildete Spalte der Tabelle

  - *label* - Beschreibung (Not yet used)
  - *row* - Abgebildete Zeile der Tabelle
  - *value* - Vom Benutzer editierbare Zahl (Standard 0)

- Cell Readonly - Repräsentiert eine *readonly* Zelle einer Tabelle mit einem Aus- und Eingang.
  - *backgroundColor* - Hintergrundfarbe
  - *column* - Abgebildete Spalte der Tabelle
  - *label* - Beschreibung (Not yet used)
  - *row* - Abgebildete Zeile der Tabelle
  - *value* - Aktueller *readonly* Wert der am Eingang anliegt (Standard 0).

- Cell Label - Eine Zelle ohne Ein- bzw. Ausgänge die vom Benutzer mit einem Text versehen werden kann.
  - *backgroundColor* - Hintergrundfarbe
  - *column* - Abgebildete Spalte der Tabelle
  - *label* - Beschreibung (Not yet used)
  - *row* - Abgebildete Zeile der Tabelle
  - *value* - Vom Benutzer editierbarer Text (Standard leer).

- Connections - Stellt eine Beziehung zwischen einem Aus- und Eingang her.

3.4.2 End-User Schnittstelle

Als End-User Schnittstelle dient eine in Eclipse eingebettete einfache tabellarische Ansicht. Die darzustellenden Werte werden aus den drei Objekten *Cell Value*, *Cell Readonly* und *Cell Label* ausgelesen. Alle Zellen können editiert werden, sofern dies für den jeweiligen Objekttyp vorgesehen ist. Weiterhin können bei allen abgebildeten Zellen die Farbe geändert werden.

Die Zellen werden automatisch synchronisiert. Finden sich jedoch Konflikte der Zellen und Spalten Positionierung, so werden diese Zellen nicht dargestellt und eine entsprechende Warnung erscheint in der Konsole, sofern diese verfügbar ist.

3.5 Fazit

Der gegenwärtige Prototyp ist unvollständig und fehleranfällig. Die Einarbeitung des Themas und der Frameworks haben zuviel Zeit gekostet um eine anständige Anwendung innerhalb der Projektzeit zu erstellen. Die Arbeit war jedoch sehr aufschlussreich, denn sie gibt Anreize und zeigt auf, in welche Schwerpunkte man sich vertiefen kann und welche für die Masterarbeit interessant sind. Um einige zu nennen:

- Entwicklung einer universellen Notation, aufbauend auf UML
- Usability-Aspekt bei der Modellierung
- End-User Schnittstellen

Literaturverzeichnis

[Alice] : *Alice - An Educational Software that teaches students computer programming in a 3D environment*. – URL http://www.alice.org/. – Zugriffsdatum: 28.09.2009

[Buxton 2009] Buxton, Bill: *Gesture based interaction*. Mai 2009. – URL http://www.billbuxton.com/input14.Gesture.pdf. – Zugriffsdatum: 28.09.2009

[Croquet] : *The Croquet Consortium*. – URL http://croquetconsortium.com/. – Zugriffsdatum: 28.09.2009

[Cypher 1993] Cypher, Allen: *Watch What I Do: Programming by Demonstration*. 1993. – URL http://acypher.com/wwid/. – Zugriffsdatum: 28.09.2009

[EUD Net] *Network of Excellence on End User Development*. – URL http://giove.cnuce.cnr.it/EUD-NET. – Zugriffsdatum: 28.09.2009. – Homepage nicht verfügbar

[EUDISMES] : *End User Development in Small and Medium Enterprise Software Systems*. – URL http://www.eudismes.de/. – Zugriffsdatum: 28.09.2009

[EUSES] : *EUSES - End Users Shaping Effective Software*. http://eusesconsortium.org/. – URL http://eusesconsortium.org/. – Zugriffsdatum: 28.09.2009

[Graphical Editor Framework] : *Graphical Editor Framework*. – URL http://www.eclipse.org/gef/. – Zugriffsdatum: 28.09.2009

[Green und Petre 1996] Green, T. R. G. ; Petre, M.: *Usability Analysis of Visual Programming Environments*. 1996. – URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.47.4836. – Zugriffsdatum: 28.09.2009

[Halbert 1984] Halbert, Daniel C.: *Programming by Example*. 11 1984. – URL http://www.halwitz.org/halbert/pbe.pdf. – Zugriffsdatum: 28.09.2009

[Hudak 1998] Hudak, Paul: *Modular Domain Sepcific Languages and Tools*. 1998. – URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.53.5061. – Zugriffsdatum: 28.09.2009

[Lieberman 2001] LIEBERMAN, Henry: *Your Wish is My Command: Programming by Example*. 2001. – URL http://web.media.mit.edu/~lieber/PBE/Your-Wish/. – Zugriffsdatum: 28.09.2009

[Lieberman u.a. 2006] LIEBERMAN, Henry (Hrsg.) ; PATERNÓ, Fabio (Hrsg.) ; KLANN, Markus (Hrsg.) ; WULF, Volker (Hrsg.): *End-User Development: An Emerging Paradigm*. 2006. – URL http://www.uni-siegen.de/fb5/wirtschaftsinformatik/paper/2006/liebermanpaternoklannwulf_enduserdevelopmentan_emergingparadigm_2006.pdf. – Zugriffsdatum: 28.09.2009

[Moore u. a.] MOORE, Bill ; DEAN, David ; GERBER, Anna ; WAGENKNECHT, Gunnar ; VANDERHEYDEN, Philippe: *Eclipse Development using the Graphical Editing Framework and the Eclipse Modeling Framework*

[Morin und Brown 1999] MORIN, Rich ; BROWN, Vicki: *Scripting Languages*. 1999. – URL http://www.mactech.com/articles/mactech/Vol.15/15.09/ScriptingLanguages/index.html. – Zugriffsdatum: 28.09.2009

[Myers 1998] MYERS, Brad A.: Natural Programming: Project Overview and Proposal / Carnegie Mellon University School of Computer. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.38.9208. – Zugriffsdatum: 28.09.2009, 1998. – Forschungsbericht

[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

[Mylyn] : *Mylyn - A task-focused interface for Eclipse*. – URL http://www.eclipse.org/mylyn/. – Zugriffsdatum: 28.09.2009

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

[Open Knowledge Simulation Modeling] : *Open Knowledge Simulation Modeling*. – URL http://oksimo.inm.de/. – Zugriffsdatum: 28.09.2009

[Schiffer 1996] SCHIFFER, Stefan: Visuelle Programmierung - Potential und Grenzen. In: *Beherrschung von Informationssystemen*. Heinrich C. Mayr, 1996, S. 267–286. – URL http://www.schiffer.at/publications/se-96-19/se-96-19.pdf. – Zugriffsdatum: 28.09.2009

[Sukale 2008] SUKALE, Martin: *Computergestützte Kunstprojekte - Neuere technologische Entwicklungen*. 2008. – URL http://users.informatik.haw-hamburg.de/~ubicomp/arbeiten/studien/sukale.pdf. – Zugriffsdatum: 28.09.2009

[Sutcliffe u. a. 2003] SUTCLIFFE, Alistair (Hrsg.) ; LEE, Darren (Hrsg.) ; MEHANDJIEV, Nik (Hrsg.): *Contributions, Costs and Prospects for End-User Development*. Bd. *IST-2002-8.1.2*. URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.122.7592&rep=rep1&type=pdf#page=3. – Zugriffsdatum: 28.09.2009, 2003