Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Projektbericht

Christian Stachow

Programming for Non-Programmers

Fakultät Technik und Informatik
Department Informatik

Inhaltsverzeichnis

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“ (Myers, 1998; Myers u. a., 2004; NatProg) 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. Deshalb konzentriert sich diese Ausarbeitung auf die Umsetzung des visuellen Aspekts der Thematik. Es werden Ideen, Entwürfe und ein experimenteller Prototyp vorgestellt.

Kapitel 2

Piktor - Piktogramm Editor

2.1 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.

2.2 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.

2.2.1 Komponenten

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

2.2.1.1 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 Abb. 2.1) 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.

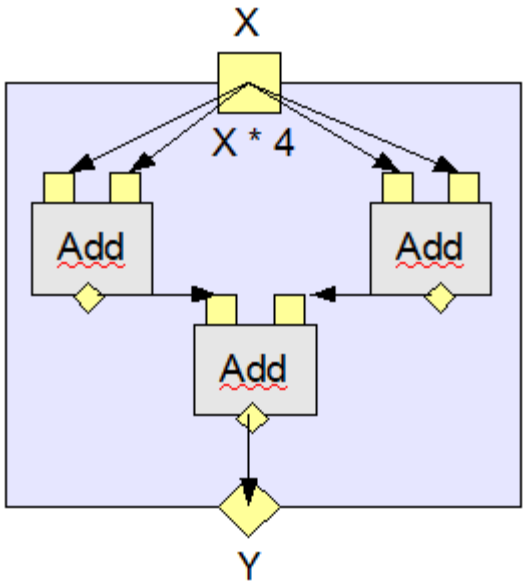

Abbildung 2.1: Blackbox Funktionsentwurf

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 $\rightarrow$ 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 2.2.2 genauer behandelt wird.

Um automatisiert Daten zu verarbeiten und die Anwendungsisolierung[1] zu vermeiden, wird ein Zugriff auf externe Quellen wie aus Datenbanken, Dateisystemen oder anderen Anwendungen (z.B. Excel) benötigt.

2.2.1.2 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 ?

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

- Anstatt eine eigene Virtuelle Maschine zu entwickeln, kann auf eine bestehende zurück gegriffen werden ?
- 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 2.2.1.3.

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[2]. 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 vier 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.
- Interpretation - Verhält sich wie virtueller Bytecode, jedoch mit dem Unterschied, dass kein zwischencode generiert wird, sondern direkt das Modell verarbeitet wird.

2.2.1.3 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

[2] Lego Mindstorm NXT ist ein kommerzielles Beispiel

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:

> 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.

2.2.2 Offene Punkte

2.2.2.1 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:

- 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)

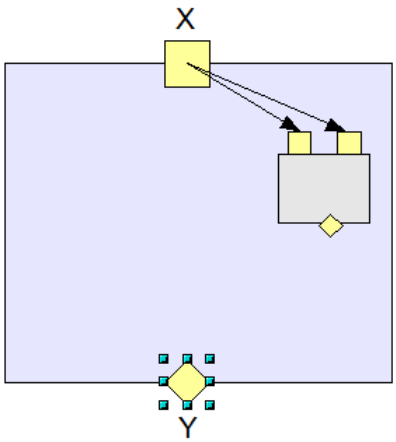

Abbildung 2.2: Skizze einer gesuchten Funktion

- 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.

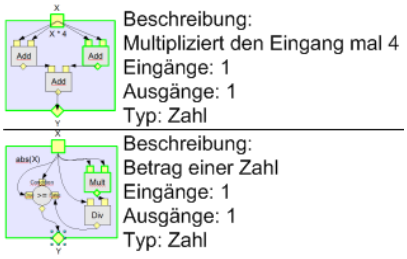

Abbildung 2.3: Trefferliste zur Skizze

2.2.2.2 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 $-a - b$

2.3 Frameworks

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

2.3.1 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.

Kurz gesagt ermöglicht EMF das grafische modellieren von Datenmodellen[3] aus denen entsprechende Java Quelltexte erstellt werden. Dieser geniererter Java Quelltext ermöglicht typische Funktionialitäten wie RUDI (Read,Update,Delete,Insert), Validation.

2.3.2 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., 2004) erklärt ausführlich die Funktionsweise von GEF mit vielen Illustrationen und Beispielen.

2.3.3 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[4] Equinox um Plug-Ins bzw. Bundles zu managen.

[3] Es existieren mehrere Möglichkeiten ein Modell aus bestehenden Quellen genieren zu lassen wie z.b. aus XSD oder annotierten Java-Interfaces

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

Das wichtigste bei einer Plug-in Entwicklung[5] 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.

2.4 Prototyp

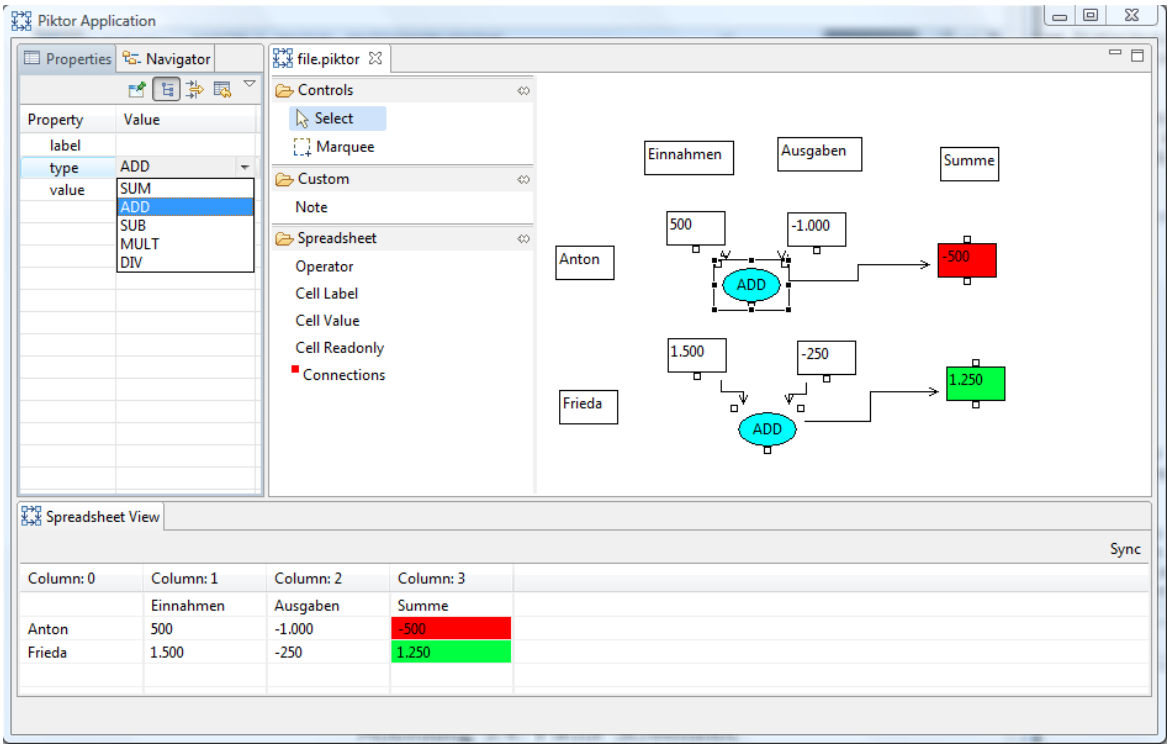

Abbildung 2.4: Piktor Screenshot

Der Prototyp namens Piktor (siehe Abb. 2.4) 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[6], 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

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

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

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 durch AOP[7] 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[8] behandelt werden. Weiterhin 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.

2.4.1 Implementierte Objekte

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

- Note - Ein Notizzettel um Beschreibungen zu platzieren

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

[8] 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.

  - *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
  - *row* - Abgebildete Zeile der Tabelle
  - *label* - Beschreibung (Not yet used)
  - *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
  - *row* - Abgebildete Zeile der Tabelle
  - *label* - Beschreibung (Not yet used)
  - *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
  - *row* - Abgebildete Zeile der Tabelle
  - *label* - Beschreibung (Not yet used)
  - *value* - Vom Benutzer editierbarer Text (Standard leer).

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

2.4.2 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.

2.4.3 Fazit

Der gegenwärtige Prototyp ist unvollständig und fehleranfällig. Die Einarbeitung des Themas, der Frameworks und einiger technischer Probleme 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 weiter vertiefen kann.

Literaturverzeichnis

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

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

[Moore u. a. 2004] MOORE, William ; DEAN, David ; GERBER, Anna ; WAGENKNECHT, Gunnar ; VANDERHEYDEN, Philippe: *Eclipse development using the graphical editing framework and the eclipse modeling framework*. Riverton, NJ, USA : IBM Corp., feb 2004. – URL http://www.redbooks.ibm.com/redbooks/pdfs/sg246302.pdf. – Zugriffsdatum: 12.10.2009. – ISBN 0738453161

[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

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

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

Abbildungsverzeichnis