Universität Hamburg
MIN-Fakultät
Fachbereich Informatik

Softwareentwicklung auf der Basis der Eclipse RCP

Am Beispiel von AVsH

Projektbericht zum Projekt 64-199
„Animationswerkzeug zur Visualisierung sozialen Handelns“
Wintersemester 2011/12

betreut von
Dr. Carola Eschenbach

vorgelegt von:
Andreas Köhn (6112777)
An.Koehn@gmail.com

März 2012

Inhaltsverzeichnis

1 Einleitung

In dem Projekt Animationswerkzeug zur Visualisierung sozialen Handelns (AVsH) wurde eine Software zum Erstellen und Verändern von Animationen mit einfachen zweidimensionalen geometrischen Objekten erstellt. Die Aufgabe des Projekts findet seine Motivation in der Psychologie und in der Mensch-Roboter-Interaktion.

Grundlegend für das Projekt ist die Attributionstheorie. Sie setzt sich mit der Untersuchung von kausalen Faktoren auseinander, die sich Menschen zur Erklärung von Effekten heranziehen (Curci-Marino, et al. 2004).

Als Begründer der Attributionstheorie gilt Fritz Heider (Curci-Marino, et al. 2004). Zusammen mit Marianne Simmel hat er im Jahr 1944 eine Studie (im Folgenden als Heider-Simmel-Studie bezeichnet) durchgeführt, in der Versuchspersonen Filme gezeigt wurden, in denen sich geometrische Figuren bewegen (vgl. Abbildung 1). Die geometrischen Figuren wurden von den Versuchspersonen, nachdem sie sich die Kurzfilme angeschaut haben, überwiegend als handelnde Lebewesen, meistens als Menschen, beschrieben.

Die Heider-Simmel-Studie hat gezeigt, dass es möglich ist, durch die Animation einfacher geometrischer Figuren die soziale Interaktion zu untersuchen und die räumliche Handlung in ihrer Dimension zu bewerten.

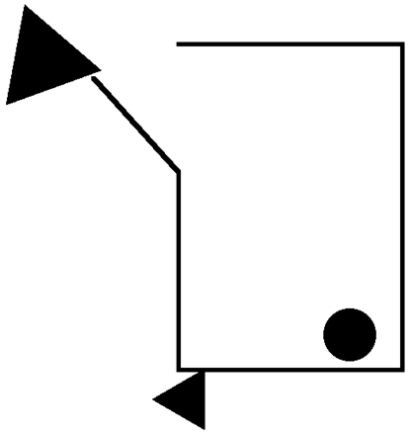

Abbildung 1: Momentaufnahme aus dem animierten Kurzfilm (Originalversuchsmaterial von Heider und Simmel)

Der Forschungsbereich Mensch-Roboter-Interaktion befasst sich unter anderem mit der Untersuchung, wie sich das Verhalten von künstlichen Objekten auf Menschen auswirkt. Es ist üblich, aufwendige Experimente mit echten Robotern in realen Umgebungen durchzuführen, um das Verhalten dieser Roboter gegenüber den Menschen zu testen. Dieses Verfahren ist mit viel Arbeit und hohen Kosten verbunden. Aufbauend auf der Heider-Simmel-Studie und im Rahmen des Projekts „Animationswerkzeug zur Visualisierung sozialer Handlungen“ ist ein Werkzeug entstanden, mit dem es möglich ist, Animationsszenen, die aus einfachen geometrischen Figuren bestehen, zu erstellen und abzuspielen. Weiterhin ist es möglich, die Animationsszenen abzuspeichern und in fast jedem gängigen Internetbrowser abzuspielen (Schäfermeier 2011).

Das Ziel dieses Werkzeugs besteht darin, schnell die Heider-Simmel-Filme zu erstellen und mit diesen Studien zu dem räumlichen Verhalten von Menschen oder auch Robotern durchzuführen.

Das Animationswerkzeug verwendet bereits existierende Bibliotheken und ist auf der Basis der Eclipse-RCP aufgebaut. Die Hauptmotivation dieses Vorgehens bestand darin, dass die Entwicklungszeit der Anwendung deutlich verkürzt wurde und der Schwerpunkt der Entwicklung auf der Funktionalität des Werkzeugs lag.

Dieser Projektbericht befasst sich mit der Entwicklung von Anwendungen auf der Basis der Eclipse Rich-Client-Platform (RCP). Dabei wird zuerst die Eclipse-RCP vorgestellt und anschließend das Animationswerkzeug aus der Sicht der RCP-Programmierung betrachtet.

2 Eclipse Rich-Client-Platform (RCP)

Das Eclipse-Projekt [1] ist hauptsächlich dafür bekannt, eine in Java geschriebene integrierte Entwicklungsumgebung (Abkürzung IDE, vom Englischen integrated development enviroment, die Eclipse-Entwicklungsumgebung wird im Folgenden als Eclipse IDE oder nur als IDE bezeichnet) zu sein. Zu dem Eclipse-Projekt gehört aber auch eine komplexe RCP.

Die Eclipse-RCP ist ein Framework zur Entwicklung modularer Desktopanwendungen, die auch als Rich-Client-Anwendung bezeichnet werden. Sie hat ihre Wurzeln in der Eclipse IDE. Viele Komponenten, auf den die Eclipse-IDE basiert, stehen auch dem RCP-Entwickler zur Verfügung, um auf der Basis dieser fertigen Komponenten eigene Anwendungen zu erstellen. Die Eclipse-RCP folgt dem modularen, Plug-In-basierten Prinzip. Ein Plug-In ist eine Erweiterung einer Anwendung um eine bestimmte Funktionalität. Ein Eclipse-Plug-In wird auch auf Basis der RCP entwickelt, wobei es dann die Eclipse IDE und Anwendungen, die auf dem RCP-Framework basieren, erweitern kann. So ermöglicht die Eclipse RCP, dass mehrere Anwendungsmodule von verschiedenen Herstellern reibungslos zusammen arbeiten, da sie alle auf dem gleichen Framework aufbauen. Rich-Client-Anwendungen vereinfachen nicht nur die Anwendungsentwicklung bzw. Weiterentwicklung durch ihren modularen Aufbau, sondern senken auch den Administrationsaufwand bei der Installation und Aktualisierung.

Im Folgenden werden die Architektur und die grafische Oberfläche der Eclipse-RCP vorgestellt. Schließlich wird das Prinzip der Plug-In-Entwicklung erläutert.

2.1 Architektur

Bis Eclipse 2.1 war Eclipse IDE monolithisch aufgebaut (Daum 2008, S. 11). Die Architektur eines monolithisch-aufgebauten Systems verbindet ihre funktionalen Elemente in einem einzigen und untrennbaren Gebilde. Die Elemente haben eine sehr enge Kopplung. Ein so aufgebautes System ist wegen der engen Kopplung sehr komplex und somit schwer zu verstehen. Die Teile des Systems können nur mit einem erheblichen Aufwand angepasst oder ausgetauscht werden, um die veränderten Anforderungen zu erfüllen. Die Einzelteile der Anwendung können nicht wiederverwendet werden (Vogel, et al. 2008). Diese Nachteile haben dazu geführt, dass für die Version 3 die Eclipse IDE in eine modular-aufgebaute Anwendung, die genau das Gegenteil eines monolithisch-aufgebauten Systems bildet, umgeschrieben wurde. Dadurch wurde auch die Ablaufumgebung geändert. Sie wurde durch eine OSGi-

[1] www.eclipse.org

Platform ersetzt. Die OSGi-Platform ist eine für Java-Module standardisierte Ablaufsumgebung, wodurch die Module im laufenden Betrieb ausgetauscht werden können (Daum 2008, S. 12). Diese Module werden auch Bundles genannt. Sie setzen sich aus den Java-Klassen und den Ressourcen zusammen.

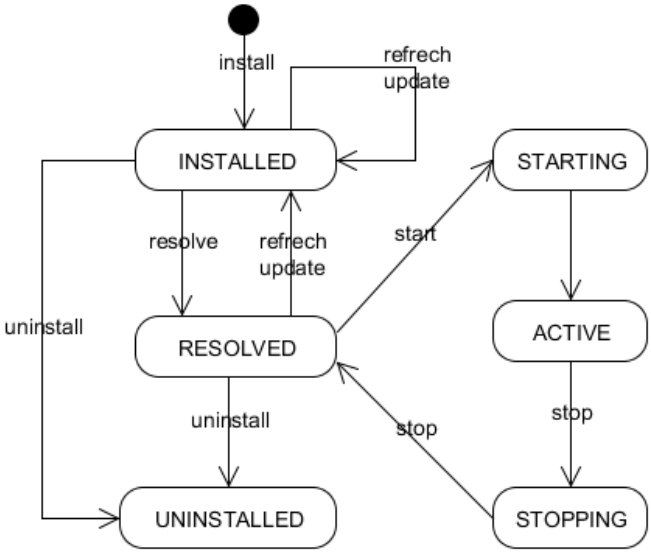

Abbildung 2: Lebenszyklus eines Bundles (Fernuni-Hagen 2010)

Der OSGi-Server verwaltet dabei die Abhängigkeiten zwischen Bundles in einer dynamischen und skalierbaren Ablaufumgebung. Die Bundles werden in einer JAR-Datei in einem OSGi-Server deponiert. Der OSGi-Server kann die Bundles installieren und entfernen. Für den Start einer Anwendung wertet der OSGi-Server den Bundle-Activator-Header der MANIFEST.MF aus, der den Namen der zu startenden Klasse enthält. Abbildung 2 stellt die Zustände, die ein Bundle im Laufe seines Lebens durchläuft, dar. In dem Zustand INSTALLED wird das Bundle installiert. Dann kann es entweder entfernt werden (UNINSTALLED) oder alle seine Abhängigkeiten werden aufgelöst (RESOLVED). In diesem Zustand kann das Bundle gestartet werden (STARTING), sodass es aktiv ist (ACTIVE). Schließlich wird er gestoppt (STOPPING) und kann entweder entfernt, neugestartet oder zuerst aktualisiert und dann neugestartet werden.

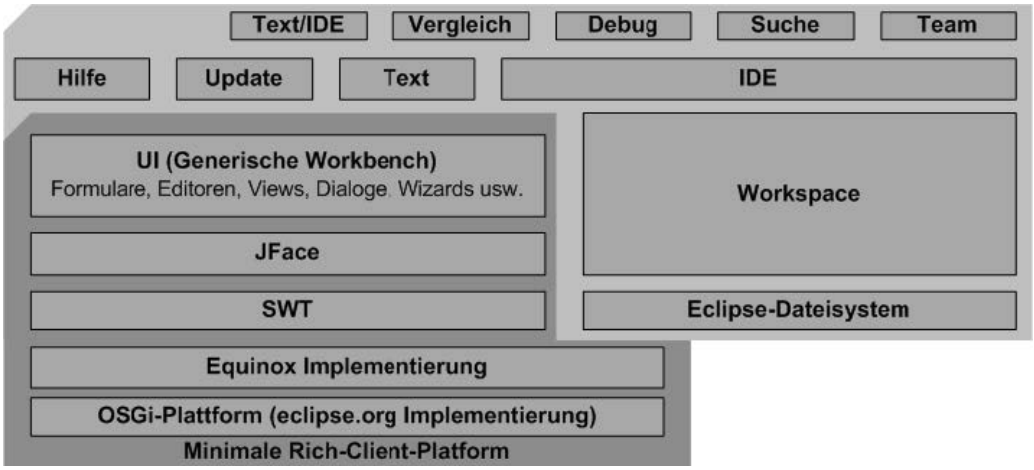

Abbildung 3: Architektur der Eclipse 3.3 (aus Daum 2008)

In Abbildung 3 ist eine Übersicht der Eclipse-Platform in der Version 3.3 dargestellt. Die dunkelgrau markierten Komponenten sind die wesentlichen Bestandteile für die Eclipse RCP. Dabei handelt es sich überwiegend um die Plug-Ins für die Gestaltung einer grafischen Benutzeroberfläche (GUI). Die Komponenten „Hilfe“, „Update“ und „Text“ sind zwar nicht Bestandteil der RCP-Minimaldistribution, können aber bei Bedarf leicht der RCP-Anwendung hinzugefügt werden, da sie die IDE-Funktionalität nicht benötigen (Daum 2008, S. 12)

2.2 Grafische Oberfläche

Als GUI-Framework verwendet Eclipse das eigens entwickelte Framework Standart Widget Toolkit (SWT). SWT nutzt zur Darstellung der Benutzeroberfläche betriebssystemspezifische Aufrufe. Dadurch passt es sich perfekt dem jeweiligen Betriebssystem an und ist somit kaum von einer nativen Anwendung unterscheidbar. Die Plattformunabhängigkeit des SWT-Toolkits wird dadurch ermöglicht, indem für sehr viele Betriebssysteme und Rechnerarchitekturen entsprechende Plug-Ins bereitgestellt werden.

Das Komponentenmodell, das von SWT zur Verfügung gestellt wird ist auf einem zu tiefen Abstraktionsniveau (Sippel, Bendisposto und Jastram 2008, S. 81), sodass der Entwickler dazu gezwungen wird, den gleichen Code an vielen Stellen zu schreiben. Dieses Problem wird durch das JFace-Framework gelöst. Das JFace-Framework setzt aus den von SWT gelieferten Basiskomponenten komplexere Widgets zusammen und stellt eine Abstraktionsschicht für den Zugriff auf die Komponenten bereit. Allerdings wird SWT nicht vollständig von JFace überdeckt (Sippel, Bendisposto und Jastram 2008, S. 81), sodass bei der Entwicklung von RCP-Anwendungen beide Frameworks benötigt werden.

Im Weiteren werden die wichtigen Komponenten der grafischen Oberfläche einer RCP-Anwendung vorgestellt.

2.2.1 Generische Workbench

Als Workbench wird der äußere Rahmen einer Eclipse RCP-Anwendung bezeichnet. Die Workbench bildet den Rahmen für alle inneren Komponenten und ist für die intelligente Positionierung der inneren Komponenten zuständig, wobei der grundlegende Aufbau durch Eclipse-RCP vorgegeben ist und durch die Anwendung-Plug-Ins lediglich erweitert wird. Der Anwender hat die Möglichkeit durch Drag&Drop diese Komponenten annährend beliebig zu positionieren. Im wesentlichen sind das die Views und Editoren, aber auch die Toolbar kann verschoben werden.

Die Workbench ist die Wurzel der GUI-Hierarchie einer Eclipse RCP-Anwendung, hat selbst aber keine visuelle Repräsentation. Stattdessen besitzt sie ein oder mehrere Fenster (Sippel, Bendisposto und Jastram 2008, S. 93). Beim Start der Eclipse RCP-Anwendung wird maximal nur eine Workbench erzeugt und gestartet, die ihre Fenster verwaltet. Nach dem Schließen des letzten Fensters wird automatisch die Workbench geschlossen (Sippel, Bendisposto und Jastram 2008, S. 93).

2.2.2 ActionBars

Eine Actionbar ist ein GUI Element, welches aus mindestens einem graphischen Element (Text/Icon), dem sogenannten Item, besteht. Durch Aktivieren des Items wird eine bestimmte Aktion ausgelöst. Innerhalb der Eclipse-RCP gibt es drei Formen von Aktionbars: Menubar, Toolbar und ein Coolbar.

In dem Menubar sind alle Items als Text dargestellt. Zusätzlich können optional erklärende Icons vorhanden sein und es werden die Tastenkürzel angezeigt, sofern die für eine Aktion definiert sind. Es gibt die Möglichkeit, dass die Editoren und Views eine Aktion an ein bestimmtes Aktionselement binden. Beim Auslösen dieses Aktionselementes bestimmt der ausgewählte Editor bzw. die View ein konkretes Verhalten wie er bzw. sie auf diese Aktion reagieren soll. Diese Aktionselemente werden als globale Aktionen bzw. „retargable Actions“ bezeichnet[1].

Eine weitere Aktionsleiste ist der Toolbar. In ihm werden die Items in Form von Icons dargestellt. Ein Toolbar kann sowohl horizontal als auch vertikal ausgerichtet sein.

Im Gegensatz dazu ist ein Coolbar eine Aktionsleiste, die nur horizontal ausgerichtet sein kann. Sie kann aus mehreren Zeilen bestehen und der Nutzer hat die Möglichkeit, die auf der Coolbar vorhandenen Items frei zu positionieren. Für Erweiterbarkeit und Funktion gelten die Aussagen für die Toolbar.

2.2.3 Views

Views stellen in einem Teilbereich des Fensters der Workbench Informationen strukturiert dar. Sie können in der Regel nur einmal in jeder Perspektive geöffnet sein. Views können die Informationen in einer Tabelle, einem Textfenster oder einem Baum darstellen. Jedoch ist es auch möglich, die Views zur Manipulation von Objekten bzw. Daten zu verwenden, zum Beispiel eine View, die die Eigenschaften eines Objektes anzeigt, erlaubt aber auch diese Eigenschaften zu ändern. Die Views können auch lokale Menüs oder Toolbars besitzen und es ist möglich, die Views durch den Nutzer zu verstecken oder wieder anzuzeigen.

2.2.4 Editoren

Ein Editor ist für die Manipulation von Texten oder Grafiken zuständig. Dabei werden die Dokumente für die Nutzer mit Editoren abgebildet. Ein Editor kann aber auch nur eine Anzeige sein und eine View zum Editieren verwenden. Dem Editor kann kein fester Platz zugeordnet werden, sondern ihm wird der restliche freie Platz in der Mitte der Workbench, den die Views nicht einnehmen, zugeordnet. Bei Bedarf kann der Benutzer dem Editor auch den vollen Bereich der Workbench zuweisen. Die Views werden dabei minimiert und bleiben solange geschlossen bis der Anwender wieder in den normalen Modus zurückwechselt.

2.2.5 Perspektiven

Perspektiven sind Container für Views und Editoren, die eine mögliche Anordnung für diese Elemente in der Workbench beschreiben. Beim ersten Starten der Anwendung existiert eine vorher definierte Anordnung zwischen den Views und dem Editor. Diese initiale Anordnung ist an die Bedürfnisse des Nutzers angepasst und wird persistent gespeichert. Der Benutzer kann jedoch die Anordnung durch Drag&Drop ändern und anpassen und, falls die Funktion bei der Anwendung implementiert ist, die Änderungen auch persistent speichern.

Es gibt aber einige Einschränkungen für den Anwender bezüglich der Anpassung der Perspektiven. Die Editoren sind nicht an eine Perspektive gebunden. Wird eine Perspektive gewechselt, so ändern sich die geöffneten Editoren nicht. Wird ein Editor

[1] http://help.eclipse.org

geschlossen, so ist er in allen Perspektiven geschlossen. Die Views sind jedoch an die Perspektive gebunden. Der Anwender hat somit die Möglichkeit für jede Perspektive zu definieren welche View sichtbar sein soll und welche nicht (McAffer, Lemieux und Aniszczyk 2010). Wird eine View in einer Perspektive geschlossen, so hat es keine Auswirkungen auf die anderen definierten Perspektiven, wo sie auch weiterhin sichtbar ist. In der Workbench ist ein Bereich entweder für eine View oder einen Editor reserviert. Der Benutzer kann nicht die View- und Editor-Reiter miteinander mischen. Für Editoren gibt es in der Workbench nur einen zusammenhängenden Bereich. Die Views hingegen können an verschiedenen Stellen gleichzeitig voneinander getrennt platziert werden (zum Beispiel eine View am linken Rand des Fensters, eine weitere am rechten Rand). Es gibt auch eine Möglichkeit, die Views aus der Workbench in einem Rahmen herauszunehmen (Sippel, Bendisposto und Jastram 2008, S. 172).

2.3 RCP Entwicklung mittels Plug-Ins

Aus der Sicht von Eclipse besteht eine RCP-Anwendung aus sehr vielen Plug-Ins. Für eine RCP-Anwendung werden mindestens ein Plug-In, eine Perspektive und ein sogenannter WorkbenchAdvisor benötigt. Der WorkbenchAdvisor steuert das Aussehen der Anwendung, also den Aufbau des Menüs, der Toolbar und der Perspektiven.

Ein Plug-In ist ein Softwaremodul, das von einer Anwendung oder einem weiteren Plug-In während der Laufzeit eingebunden werden kann um die Funktionalität der Anwendung oder des Plug-Ins zu erweitern. Es ist wichtig, dass ein Plug-In eine klar definierte Schnittstellenfunktionalität bereitstellt. Dabei besteht ein Plug-In einer RCP-Anwendung mindestens aus zwei XML-Dateien, der MANIFEST.MF, die sich in dem Ordner META-INF befindet, und der plugin.xml, die sich direkt in dem Projektordner befindet. Damit hätte das Plug-In nur einen deklarativen Charakter. Um weitere Funktionalitäten unterzubringen, müssen zusätzliche Java-Klassen implementiert werden (Daum 2008, S. 66).

Die Datei MANIFEST.MF ist das OSGi-Manifest. Es beschreibt die Abhängigkeiten zwischen dem Plug-In, zu dem dieses Manifest gehört, gegenüber anderen Plug-Ins. Außerdem beschreibt es, welche eigene Pakete gegenüber anderen Plug-Ins öffentlich sichtbar sein sollen.

Neben der Datei MANIFEST.MF wird das Plug-In-Manifest in der plugin.xml hinterlegt. In dieser Datei benennt das Plug-In-Manifest die Erweiterungspunkte (Extension Points) des Plug-Ins. Diese stellen Erweiterungsschnittstellen dar, die von anderen Plug-Ins verwendet werden können. Die Plug-Ins, die die Funktionalität von den anderen Plug-Ins erweitern sollen, müssen dazu passende Extensions (Erweiterungen) in der Datei plugin.xml deklarieren. Zusätzlich können diese auch neue Erweiterungspunkte definieren, welche wiederum von anderen Plug-Ins als Erweiterungsschnittstellen genutzt werden können, was in der Abbildung 4 grafisch dargestellt ist. Die Datei plugin.xml beschreibt somit „eine Art Schlüssel-Schloss-Prinzip“ (Holstein 2007, S. 11). Dieses Prinzip ermöglicht es, eine komplexe Anwendung aus einer Menge von Plug-Ins zu erstellen. Auch die Bestandteile der RCP-Architektur aus dem Abschnitt 2.1 Architektur sind nichts anderes als Plug-Ins, die in jeder RCP-Anwendung zu finden sind.

Eclipse bietet einen komfortablen, grafischen Plug-In Manifest Editor zum Verändern der beiden XML-Dateien, sodass man nur selten direkt mit den XML-Dateien arbeiten muss. Dieser Editor wird im Teil 3, „Der Animationswerkzeug-Plug-In“, an dem Beispiel des Animationswerkzeugs näher erläutert.

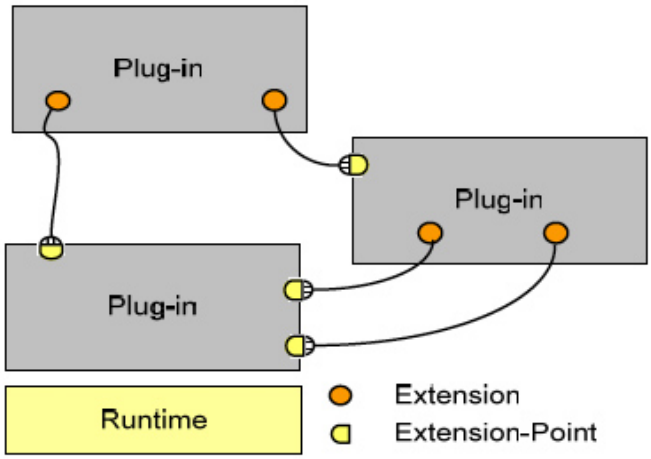

Abbildung 4: Verwendung von Extension und Extension Points (Schaare, S. 3)

Wie schon oben erwähnt, basiert die Eclipse-RCP auf der OSGi-Platform. In Eclipse wird die Aufgabe von dem OSGi-Server, von den Plug-Ins org.eclipse.osgi und org.eclipse.core.runtime übernommen. Die zwei Plug-Ins bilden eine Laufzeitumgebung (Runtime) (siehe auch Abbildung 4), die den Lebenszyklus eines jeden Plug-Ins verwaltet. Die Plug-Ins einer Anwendung sind den Bundles von OSGi gleichgestellt. Die interne Realisierung von Eclipse bildet die Plug-Ins auf OSGi-konforme Bundles ab. Die Voraussetzung dafür ist, dass jedes Plug-In eine Instanz des Interfaces BundleActivator sein muss und eine Manifest-Datei, die zu jedem Plug-In gehört, bereitstellen muss.

Plug-Ins werden erst in den Speicher geladen, wenn ihre Funktionalität benötigt wird (Lazy-Loading). Der Vorteil davon ist, dass man die Anwendungen mit theoretisch unbegrenzter Anzahl von Plug-Ins erweitern kann, da die installierte Erweiterung zunächst so gut wie keine Ressourcen benötigt

3 Der Animationswerkzeug-Plug-In

Das Animationswerkzeug zur Visualisierung sozialen Handelns wurde auf der Basis der Eclipse-RCP entwickelt und stellt daher einen Plug-In, der durch die Anbindung von anderen Plug-Ins eine komplexe Anwendung bildet, dar.

In diesem Kapitel wird das Animationswerkzeug aus der Sicht der Eclipse-RCP betrachtet. Dabei wird zuerst auf die Grundlagen, die für das Verstehen der Funktionen des Animationswerkzeugs notwendig sind, eingegangen. Als nächstes wird der Plug-In Manifest Editor am Beispiel der Anwendung erläutert und zum Schluss werden die Plug-Ins der grafischen Oberfläche vorgestellt.

3.1 Grundlagen

3.1.1 SVG

Scalable Vector Graphics (SVG) ist eine Spezifikation basierend auf XML zur Beschreibung zweidimensionaler Vektorgrafiken. SVG ermöglicht es mittels XML verlustfreie Vektorgrafiken zu entwickeln, die zudem noch eine sehr kleine Dateigröße haben, da sie nur aus Text bestehen und in fast allen Browsern dargestellt werden können.

Ein weiterer Vorteil von SVG ist, dass mit seiner Hilfe sich verschiedene Animationen auf der Basis der 2D-Grafiken erzeugen lassen. Das geschieht durch die eingebettete

Synchronized Multimedia Integration Language (SMIL) Spezifikation, die ebenso XML-basiert ist und direkt in die Beschreibung der SVG-Elemente einfließen kann (Röhling 2011).

3.1.2 GEF

Das Grafik Editing Framework (GEF) ist ein open source Java-Framework, das eine große Bandbreite an Funktionalität für die Entwicklung von Anwendungen mit einem grafischen Editor zur Erstellung von 2D-Figuren bietet.

GEF setzt auf der Eclipse-RCP auf und besteht aus zwei Plug-Ins: Draw2d und GEF. Die Abbildung 5 stellt die Abhängigkeit zwischen den zwei Plug-Ins und der Eclipse-RCP dar.

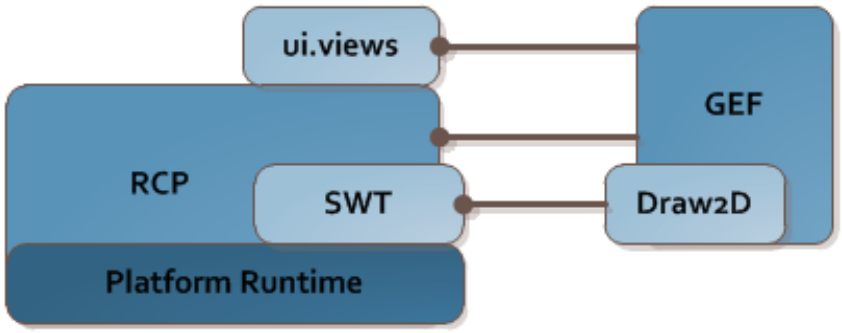

Abbildung 5: GEF - Abhängigkeiten der Plug-Ins (Röhling 2011)

Das Plug-In Draw2d baut auf SWT auf und stellt einen leichtgewichtigen Überbau (Daum 2008) über diese Bibliothek dar, was den Umgang mit den von SWT benutzten Betriebssystemkomponenten sehr stark vereinfacht. Anstelle direkt mit diesem arbeiten zu müssen, kann man mit Hilfe von Draw2d direkt mit grafischen Objekten (z.B. Panel, Rechteck oder Hilfslinien) hantieren (Daum 2008, S. 458).

Das Plug-In GEF ermöglicht es, die grafischen Objekte zu editieren. Dabei basiert es auf dem Model-View-Controller-Pattern. Die View wird von Draw2d-Figuren und Connections dargestellt. Das Model stellt der Programmierer zur Verfügung und das Plug-In GEF übernimmt den Part des Controllers (Metzger und Kunz 2005).

3.2 Der Plug-In Manifest Editor

Bei der Bearbeitung des deklarativen Teils eines Plug-Ins benutzt man üblicherweise den Plug-In-Manifest-Editor, wie in Abbildung 6 dargestellt. Der Editor wird von der Eclipse IDE zur Verfügung gestellt. Um ihn zu starten reicht es, die Datei plugin.xml zu öffnen. Der Vorteil zu der manuellen Bearbeitung der einzelnen XML-Dateien ist, dass durch die Änderung in dem Editor gleich mehrere Dateien bearbeitet und keine XML-Kenntnisse benötigt werden. Der Editor bietet somit eine einheitliche Sicht auf die Dateien: plugin.xml, MANIFEST.MF und die build.properties.

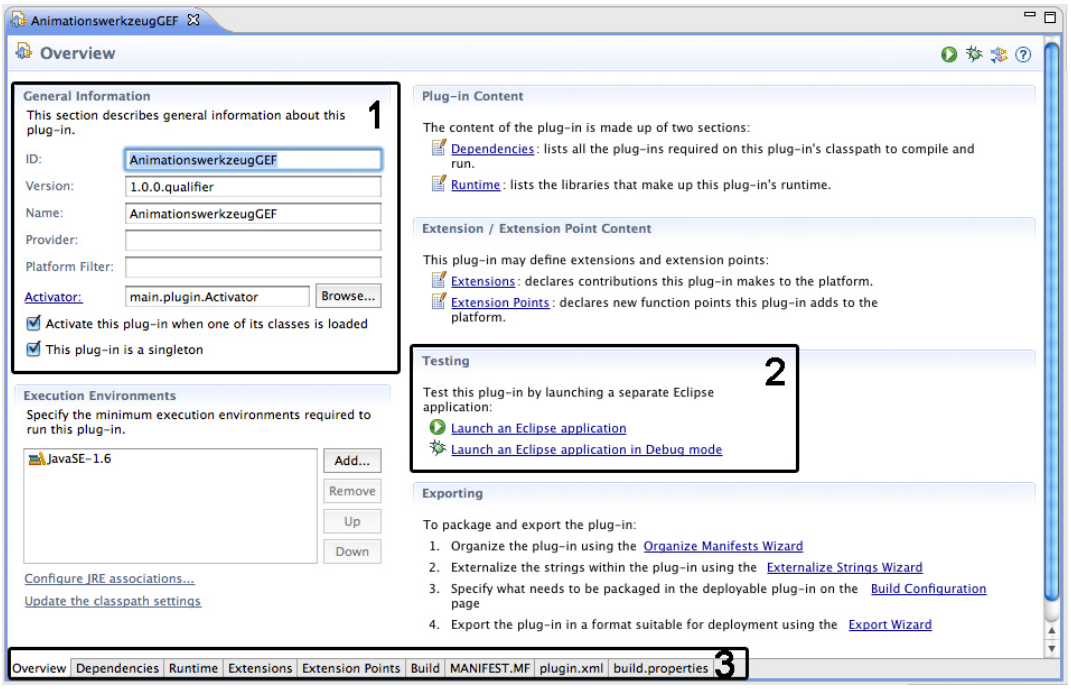

Abbildung 6: Übersichtsseite des Plug-In Manifest Editors von dem Animationswerkzeug

Die Abbildung 6 zeigt die Übersichtsseite des Editors. Oben links, unter 1, befindet sich die allgemeine Information wie die ID, Version und Name des Plug-Ins. Außerdem muss im Feld Activator die Java-Klasse eingetragen werden, die den Lebenszyklus des Animationswerkzeugs steuert. Bei dem Animationswerkzeug ist die Klasse main.plugin.Activator dafür zuständig. Weiterhin wird in diesem Bereich festgelegt, ob für das Plug-In ein Lazy-Loading verwendet werden soll, oder nicht.

Über den Bereich Testing (2) wird die Anwendung gestartet. Dabei gibt es auch die Möglichkeit, die Anwendung im Debug-Modus zu starten.

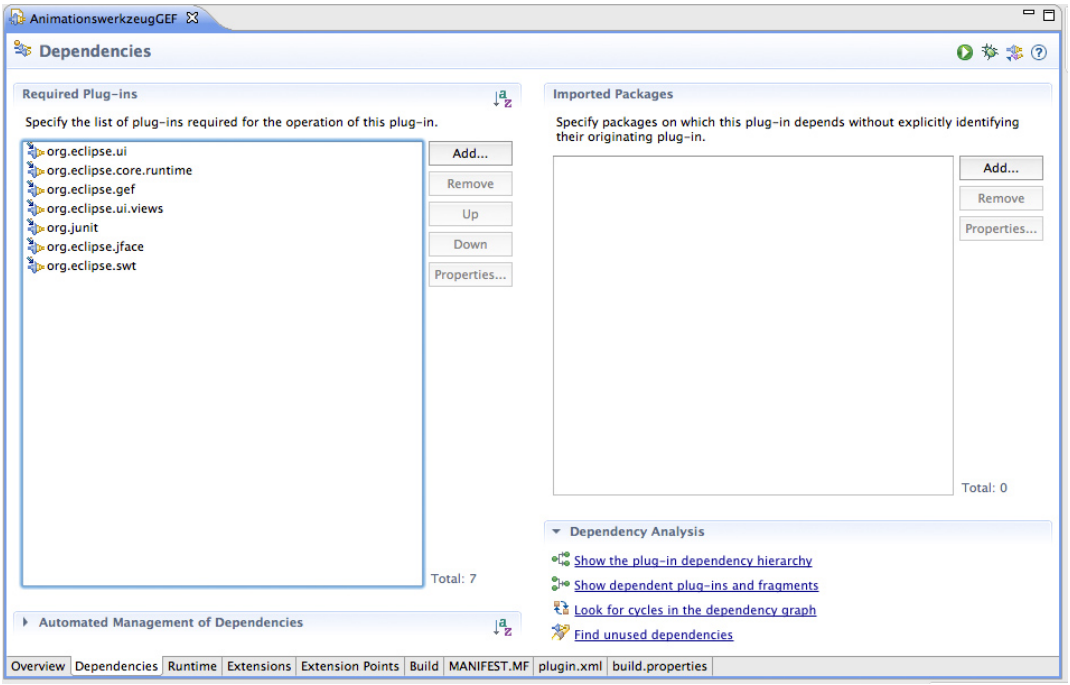

Abbildung 7: Dependencies-Seite des Plug-In Manifest Editors von dem Animationswerkzeug

Über die untere Leiste (3) gelangt man zu den anderen Seiten des Editors. Über die letzten drei Seiten MANIFEST.MF, plugin.xml und build.properties lassen sich diese Dateien manuell konfigurieren.

Auf der Dependencies-Seite (vgl. Abbildung 7) werden die Abhängigkeiten zu anderen Plug-Ins (linke Seite) bzw. Paket-Importen (rechte Seite des Editors) deklariert. Der Unterschied dabei ist, dass falls ein Paket importiert wird, seine Herkunft die Anwendung nicht interessiert. Jedes beliebige Plug-In, das das entsprechende Paket exportiert, kann benutzt werden. Wird hingegen die Abhängigkeit zu einem bestimmten Plug-In deklariert, so können alle Pakete, die dieses Plug-In exportiert, benutzt werden. Beide Methoden haben ihre Vor- und Nachteile. Während der Import flexibler ist, ist bei der Verwendung einer Abhängigkeit besser vorherzusehen, was passiert (Sippel, Bendisposto und Jastram 2008). Das Animationswerkzeug deklariert nur die Abhängigkeiten zu den Plug-Ins, unter denen sich auch zum Bespiel GEF befindet und kann somit alle exportierten Pakete dieser Plug-Ins nutzen.

Die Runtime-Seite deklariert die Klassen, welche für andere Plug-Ins sichtbar sein sollen. Dabei kann man Packages nur für bestimmte Plug-Ins sichtbar machen. Außerdem werden hier die Laufzeit-Klassenpfade festgelegt. Beim Animationswerkzeug sind die zwei Bibliotheken Batik und Xuggle aus dem Projekt in ein separates Projekt AWLibraries rausgenommen. Der Grund dafür ist, dass falls die Bibliotheken in dem gleichen Projekt sind, werden sie ständig nach Fehlern überprüft. Da die Klassen in den Bibliotheken nicht verändert werden, kostet es viel Zeit sie ständig zu überprüfen. Deswegen wurden sie verlagert. Weil die Bibliotheken sonst für das Animationswerkzeug nicht sichtbar sind, müssen die Laufzeit-Klassenpfade von Batik und Xuggle in Runtime festgelegt werden.

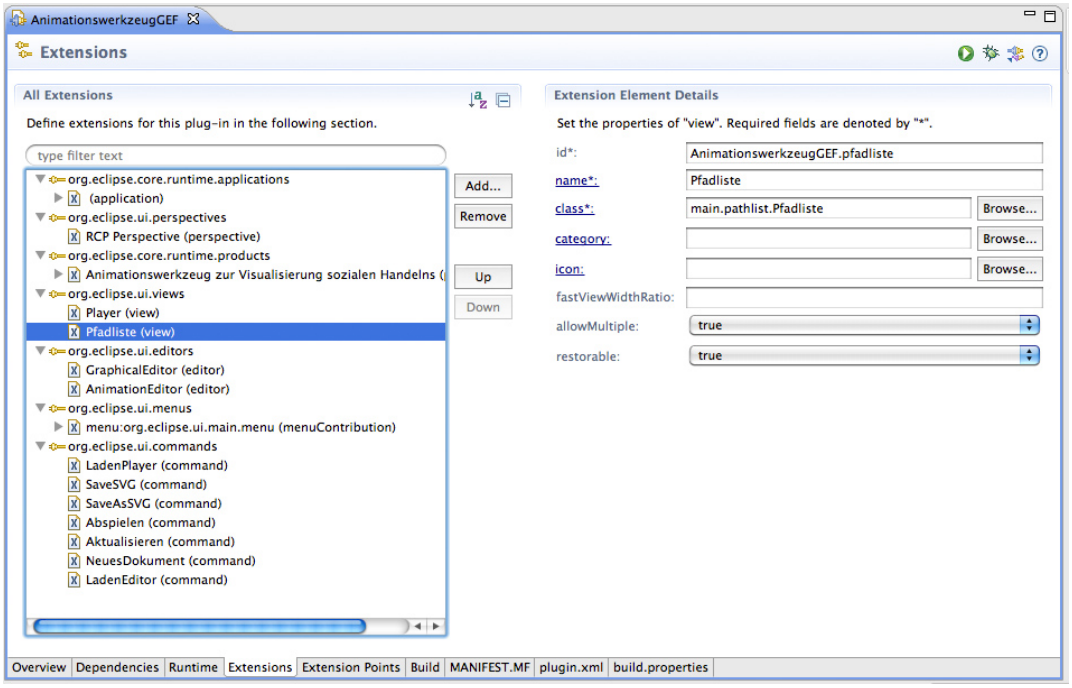

Abbildung 8: Deklarationsseite für Erweiterungen des Plug-In Manifest Editors von dem Animationswerkzeug

Die Extension-Seite, die in Abbildung 8 dargestellt ist, ist für das Erweiterungskonzept sehr wichtig. Hier wird deklariert welche Erweiterungs-Plug-Ins die Anwendung zur Verfügung hat und die „losen“ Plug-Ins werden zu einer Anwendung verankert. Links in dem Editor werden die Elemente der Erweiterungen in einer Baumstruktur dargestellt. Aus Abbildung 8 erkennt man, dass zum Zeitpunkt der Erstellung dieses

Projektberichtes das Animationswerkzeug nur die Plug-Ins für die grafische Oberfläche als Erweiterungen hat. In dem rechten Teil der Extensionsseite befindet sich der Detaileditor, der es ermöglicht die Attribute eines bestimmten Elementes zu ändern. Die Anzahl und der Typ der einzelnen Attribute hängt von dem Element ab. Bei meisten Elementen muss man aber die ID, Name und die Java-Klasse, die den Plug-In implementiert, angeben.

3.3 Grafische Oberfläche

Die Abbildung 9 stellt die Workbench des Animationswerkzeugs in der Explosionsansicht dar. Folgende Komponenten beinhaltet die Workbench zum Zeitpunkt der Erstellung dieses Projektberichtes:

1. Menüzeile
2. Symbolzeile
3. Grafik- und Animationseditor
4. Eigenschaftentabelle
5. Pfadliste
6. Player

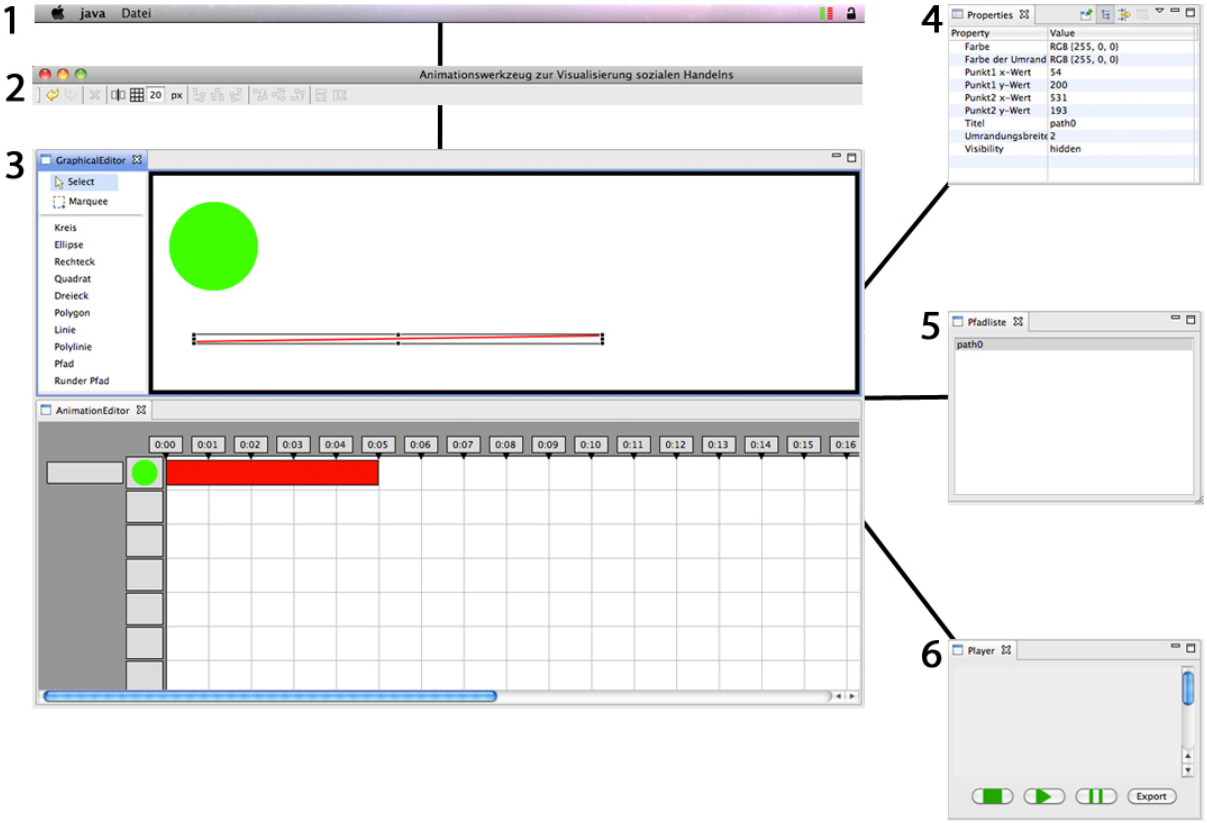

Abbildung 9: Explosionsansicht des Animationswerkzeugs

3.3.1 Actionbar

Es gibt zwei Actionbars bei dem Animationswerkzeug. Menüleiste (1) und der Toolbar (2). Die Commands für die Menüleiste werden in dem Plug-In Manifest Editor auf der Extensions-Seite in das Animationswerkzeug eingebunden (siehe auch Abbildung 8 unterorg.eclipse.ui.commands) und stellen somit die Erweiterungs-Plug-Ins der Anwendung dar. Hier sind auch die Klassen, die für die Verarbeitung der Commands zuständig sind, angegeben.

Auf dem Toolbar befinden sich die Aktionen, die von GEF zur Verfügung gestellt werden. Die Aktionselemente Undo, Redo und Delete sind globale Aktionen, die sowohl für den Grafikeditor als auch für den Animationeditor gelten. Die anderen Aktionselemente, wie Raster und Ausrichtungsaktionselemente, lösen nur bei dem Grafikeditor bestimmte Aktionen aus.

3.3.2 Editoren

Das Animationswerkzeug hat zwei Editoren (3), die auf GEF aufbauen: Grafik- und Animationseditor. Die zwei Editoren wurden über die Erweiterungsseite des Plug-In Manifest Editors an das Animationswerkzeug angebunden, erben aber von den Klassen GraphicalEditorWithPalette bzw. GrapphicalEditor, die GEF zur Verfügung stellt und übernehmen somit die Eigenschaften eines grafischen Editors, der auf GEF aufbaut.

Mit Hilfe von dem Grafikeditor ist es möglich, statische Objekte zu erzeugen. Dafür bietet er zehn verschiedene Formen von Objekten an, die jeweils über die Werkzeugpalette am linken Rand auszuwählen sind. Die Werkzeugpalette stellt einen Toolbar bei einer Grafikanwendung dar, der eine Sammlung von Aktionen, zum schnellen Editieren der grafischen Objekte im Grafikeditor, anbietet.

Der Animationseditor dient dazu Animationen wie Farb-, Bewegung-, Rotation-, Attribut- oder Rotationsanimation zu erstellen. Die Szene kann anschließend in einer SVG-Datei gespeichert werden.

3.3.3 Views

Zum Zeitpunkt der Erstellung dieses Projektberichtes besitzt das Animationswerkzeug drei Views: Eigenschaftentabelle (4), Pfadliste (5) und der Player(6).

Die Eigenschaftentabellenview ist von GEF vorgegeben. Das erkennt man zum Einen daran, dass sie nicht in der Extensions-Liste (siehe Abbildung 8) unter org.eclipse.ui.views aufgelistet ist, zum Anderen hat diese View eine vordefinierte Coolbar mit verschiedenen Aktionselementen (siehe Abbildung 9).

Die Eigenschaftentabellenview stellt die Eigenschaften einer ausgewählten Figur aus dem Grafik- bzw. einer ausgewählten Animation aus dem Animationseditor in einer Tabelle dar. Diese View erlaubt aber auch die Eigenschaften der ausgewählten Objekte zu ändern. Dieses Beispiel zeigt, dass die Views einer Eclipse-RCP Anwendung nicht nur zum Anzeigen der Daten benutzt werden können, sondern auch zum Editieren dieser Daten.

Die Pfadlisten- und Playerview wurden über die Deklarationsseite für Erweiterungen (siehe Abbildung 8) an die Anwendung angebunden. Die Pfadliste hat die Aufgabe, die in dem Grafikeditor erstellten Pfade in einer Liste anzuzeigen. Wählt man einen Pfad aus der Liste, so wird er in dem Grafikeditor ausgewählt und man kann sofort eine Bewegungsanimation in dem Animationseditor erstellen. Die Playerview spielt die erstellten Animationen ab und kann diese Animationen als Videodateien exportieren.

4 Diskussion

Durch die Entwicklung der Anwendungen auf der Basis von Eclipse-RCP erreicht man durch die Plug-In-basierte Entwicklung eine enorme Einsparung bei der Programmierung von Standart-Features. Beispiele für solche ist die standardisierte grafische Oberfläche oder auch das Framework GEF.

Des Weiteren besitzen die meisten RCP-Anwendungen eine einfache Endbenutzeranpassung (zum Beispiel mit Hilfe von Perspektiven) und können auch sehr leicht an die Nutzer und deren verschiedene Betriebssysteme verteilt werden.

Um alle Vorteile der Eclipse-RCP zu nutzen, muss man zum Anfang eine gewisse Einarbeitungszeit für die Plattformprogrammierung investieren. Ist es gemacht, kann man sehr viel Zeit bei der Implementierung für die Entwicklung einiger Besonderheiten (Staron 2011) sparen.

Die zwei Editoren des Animationswerkzeugs zur Visualisierung sozialen Handelns, die die Grundkomponenten der Anwendung darstellen, basieren jeweils auf einem GEF-Editor. Dadurch, dass GEF den Umgang mit den von SWT benutzten Betriebskomponenten stark vereinfacht, eine Menge von grafischen Objekten und einen grafischen Editor zur Verfügung stellt, konnte die erste Projektgruppe, die im Wintersemester 2010/2011 mit der Entwicklung der Anwendung angefangen hat, enorm viel Zeit bei der Entwicklung der zwei Editoren einsparen. Die Gruppe konnte die Editoren an die Ansprüche des Projektes anpassen und musste sie nicht neu entwickeln.

Die zwei Views Player und Pfadliste dienen dazu, dem Nutzer mehr Informationen über das bearbeitete Projekt darzustellen. Player kann die erstellte Szene abspielen und sie als eine Videodatei exportieren. Die Pfadliste zeigt die erstellten Pfade in einer Liste an, sodass der Nutzer die Pfade aus der Liste schneller auswählen kann. Die zwei genannten Views erlauben aber nur die Darstellung der Szene bzw. der Objekten und nicht die Manipulation dieser. Die dritte View, die Properties-View, macht beides. Sie zeigt die Eigenschaften eines ausgewählten Objektes an, erlaubt die aber auch zu ändern. Sie übernimmt also auch die Aufgabe eines Editors, der dafür da ist, die Objekte zu bearbeiten. Eine View kann durchaus eine Eigenschaft haben, die ihr erlaubt, die Manipulation der Objekte durchzuführen. Bei dem Animationswerkzeug kann die Properties-View mehr Eigenschaften eines Objektes ändern als der GraphicalEditor selber, z. B. kann die View die Farbe des Objektes ändern, der Editor aber nicht. Da ein Editor hauptsächlich dafür da ist, die Objekte zu bearbeiten, sollte eine View, meiner Meinung nach, nicht mehr Funktionen zur Manipulation als der Editor haben. Sonst kann es dazu führen, dass der Nutzer des Werkzeugs die Übersicht verlieren kann, da die Manipulationsmöglichkeiten auf verschiede Fenster verteilt wurden, sodass man sie zuerst suchen muss. Besser ist es, wenn ein Editor alle Manipulationsmöglichkeiten besitzt, die Views aber einige davon auch anbieten, falls es sinnvoll ist.

Alles in Einem kann man sagen, dass durch die Benutzung verschiedener Bibliotheken und dem Framework GEF, in einer relativ kurzer Zeit ein funktionsfähiges Werkzeug entstanden ist. Das Animationswerkzeug ist noch stark ausbaufähig. Durch die Entwicklung der Anwendung auf der Basis der Eclipse-RCP müssen sich die Studenten nicht um Grundkomponenten kümmern, sondern den Blick auf die Implementierung der Funktionen, die das Animationswerkzeug zur Visualisierung sozialen Handelns erfüllen muss, legen.

5 Quellenverzeichnis

5.1 Literaturverzeichnis

Vogel, Oliver, et al. *Software-Architektur: Grundlagen - Konzepte - Praxis.* 2. Auflage. Heidelberg: Spektrum Akademischer Verlag, 2008.

Daum, Berthold. *Java-Entwicklung mit Eclipse 3.3.* 5. Auflage. Heidelberg: dpunkt.verlag, 2008.

Daum, Berthold. *Rich-Client-Entwicklung mit Eclipse 3.3.* 3. Auflage. Heidelberg: dpunkt.verlag, 2008.

Holstein, Sven. „Rechnerverarbeitbare Gebäudemodelle für die Projektierung von Rechnernetzen innerhalb einer RCP-Umgebung." *Diplomarbeit.* Dresden, 2007.

McAffer, Jeff, Jean-Michael Lemieux, und Chris Aniszczyk. *Eclipse Rich Client Platform.* 2. überarbeitete Auflage. Amsterdam: Addison-Wesley, 2010.

Metzger, Rico, und Peter Kunz. „Eclipse Entity Relationship Diagram Editor mit Codegenerierung." Zürich: Zürcher Hochschule Winterthur, 2005.

Schaare, Glenn. „Varianten-freundliches Software-Design: Die Eclipse-Rich-Client Platform." *Seminararbeit.* Münster.

Schäfermeier, Stefan. „Animationseditor: Anforderungen & Lösungen." *Projektbericht.* Hamburg, 2011.

Sippel, Heiko, Jens Bendisposto, und Michael Jastram. *Eclipse Rich Client Platform.* 1. Auflage. Frankfurt am Main: entwickler.press, 2008.

Staron, Tobias. „Der Grafikeditor und GEF: die Umsetzung, das Modell und der Eigenschafteneditor." *Projektbericht.* Hamburg, 2011.

Röhling, Sven. „GEF als Werkzeug für Editoren." *Projektbericht.* Hamburg, 2011.

5.2 Webseiten und Poster

Curci-Marino, Loredana, Matthias Spörrle, Peter Hinterseer, und Friedrich Försterling. „Zur Personenwahrnehmung im Attributionsgeschehen: Eine Replikation der klassischen Arbeit von Heider und Simmel (1944)." *Ludwig-Maximilians-Universität München.* 04. 04 2004. http://epub.ub.uni-muenchen.de/1396/ (Zugriff am 29. 02 2012).

Fernuni-Hagen. *Plugins.* 2010. http://wiki.fernuni-hagen.de/eclipse/index.php/Plugins#Lebenszyklus (Zugriff am 08. 03 2012).