Universität Hamburg
Fakultät für Mathematik, Informatik und Naturwissenschaften
Fachbereich Informatik
Projekt: Animationswerkzeug zur Visualisierung sozialen Handelns
Leitung: Dr. Carola Eschenbach, Felix Lindner
Wintersemester 2010/2011

Evaluation der Softwarearchitektur

Zusammenfassung

In dem Projekt *Animationswerkzeug zur Visualisierung sozialen Handelns* (AVsH) wurde eine Software zum Erstellen und Verändern von Animationen mit geometrischen Objekten in einem grafischen Editor erstellt. Der Film wird im *Scalable Vector Graphics* (SVG)-Format gespeichert. SVG bietet eine standardisierte XML-Struktur, die von Browsern interpretiert werden kann. Somit ist es möglich Filme über ein SVG-Dokument zu beschreiben. Der Programmieraufwand steigt für längere Filme und die Übersichtlichkeit des Dokumentes geht verloren. Daher können mit dem Editor Dokumente unter Benutzung einer grafischen Oberfläche erstellt und überarbeitet werden.
Ein Projekt mit einer grafischen Oberfläche bedeutet großen Programmieraufwand und bedarf zur Wartung eine gute Softwarearchitektur. Mit Hilfe von Frameworks wie dem *Graphical Editing Framework* (GEF) ist es möglich die Softwarearchitektur stark zu beeinflussen, da Grundstrukturen vorgegeben sind.
In dieser Arbeit wird die Softwarearchitektur analysiert und evaluiert mit dem Ziel Vorschläge zur Verbesserung zu liefern, um so die Software-Architektur zu verbessern und das Projekt besser wartbar und erweiterbar zu machen.

Vorgelegt von:
Jens Dallmann
Matrikelnummer: 6043783
8dallman@informatik.uni-hamburg.de

Software-System-Entwicklung
3. Fachsemester

Hamburg, den 29. April 2011

Inhaltsverzeichnis

1 Einführung

Die Aufgabe des Projektes findet seine Motivation in der Mensch-Roboter-Interaktion und in der Psychologie. Gemeinsam im Team sollte ein Programm erstellt werden, um zweidimensionale animierte Szenen von primitiven geometrischen Objekten zu erzeugen. In der Vergangenheit gab es Filme dieser Art von Heider & Simmel [1], weswegen sie im Folgenden als Heider-Simmel-Filme bezeichnet werden. Heider-Simmel-Filme werden oft in Interpretationen Handlungen zugeschrieben. Probanden erklärten beim Anschauen einer Szene, dass ein Dreieck mit einem Kreis spielt, ihn verfolgt oder bedroht. Die Interpretation der Szene als Situation, mit Bezug zum Leben, wird durch die Benutzung von Attributen zur Beschreibung der Objekte deutlich.
Mit einem grafischen Editor soll die These überprüft werden, dass mit Heider-Simmel-Filmen das Verhalten von Robotern im Raum analysiert werden kann. Es geht speziell darum, dass Roboter mit uns Menschen interagieren und dabei einer Erwartung unterliegen. Wenn das Verhalten unnatürlich und unerwartet ist, fällt Menschen auf, dass etwas nicht ihren Erwartungen gemäß geschieht, und es kann ein Gefühl der Bedrohung entstehen.
Ein Editor, der für diesen Zweck entwickelt wurde und daher einfach zu bedienen ist, existierte bisher noch nicht. Das Entwickeln des Editors wurde zur Aufgabe des Projekts. In der Planungsphase wurde der Editor in vier Module unterteilt. Der Animationseditor, der Grafikeditor, der Player und das zentrale Modell.
Abb. 1 zeigt das zu Beginn ausgearbeitete Abhängigkeitsverhältnis des Programms. Der Grafikeditor und der Animationseditor erhalten Informationen aus dem zentralen Szenenmodell und können das zentrale Szenenmodell verändern. Der Player kann das zentrale Szenenmodell nicht beeinflussen, bekommt aber eine Szene zum Abspielen.

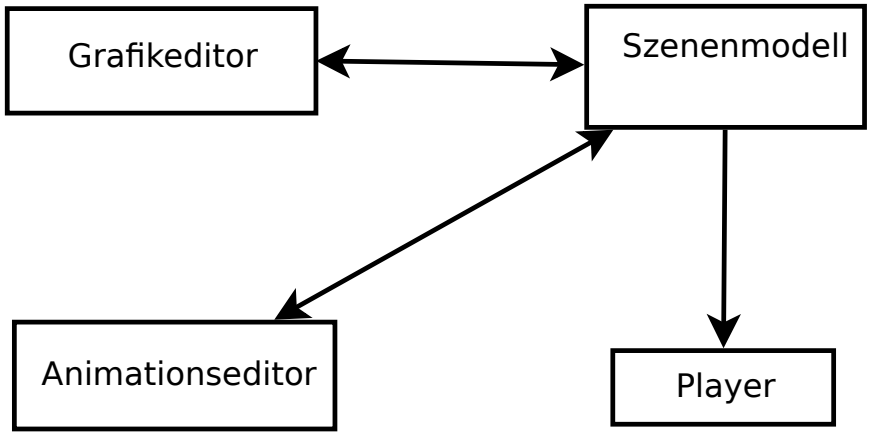


Abbildung 1: Grundlegendes Abhängigkeitsverhältnis der einzelnen Module des Projekts.

2 Design-Patterns

Design-Patterns sind mögliche Lösungen für ein Problem. In der Softwareentwicklung tauchen bestimmte Probleme immer wieder auf. Oft sind diese Probleme umständlich zu implementieren. Daher gibt es Veröffentlichungen [2] [3] von Programmierern, die eine ihrer Meinung nach gute Lösung für ein bestehendes Problem gefunden haben.
Es kann also eine Vielzahl an Lösungen für ein ähnliches Problem geben, die unterschiedliche Vor- und Nachteile mit sich bringen. Das Schwierige bei Design-Patterns ist zu erkennen, wann ihr Einsatz Sinn macht. In diesem Abschnitt werden im Projekt benutzte Design-Patterns für die Objektorientiere Programmierung vorgestellt.

2.1 Observer

Das Design-Pattern *Observer* (dt.: Beobachter) bietet eine Lösung zur Benachrichtigung von Beteiligten bei einer Veränderung. Ein Beobachter wird benachrichtigt, sobald sich der Objektzustand des Beobachteten geändert hat. Dies hat den Vorteil, dass Beobachter, die an dem Objektzustand interessiert sind, sich nicht immer wieder selbst nach dem Objektzustand erkundigen müssen. In der Regel läuft es so, dass sich die Beobachter bei einem beobachteten Objekt anmelden. Die Beobachter bekommen im Falle einer Änderung ein Ereignis mit dem neuen Objektzustand geschickt und können entscheiden, ob die Zustandsänderung relevant für sie ist.
Die Aktualisierung läuft über eine Methode eines Interfaces, das jeder Beobachter implementieren muss. Dies hat den Vorteil, dass die Objekte lose gekoppelt sind, da sie nichts über die weiteren Strukturen voneinander wissen müssen. In Abb. 2 sieht man die Kopplung der Klassen. Das Subjekt kennt Beobachter, allerdings keine konkreten Beobachter. Konkrete Beobachter implementieren das Interface Beobachter. Ein Problem des Observer-Patterns ist, dass immer alle Beobachter benachrichtigt werden müssen, da das Subjekt nicht unterscheiden kann, ob der Beobachter an den Informationen interessiert ist. Dies kann bei vielen Beobachtern zu Performanz-Problemen führen.

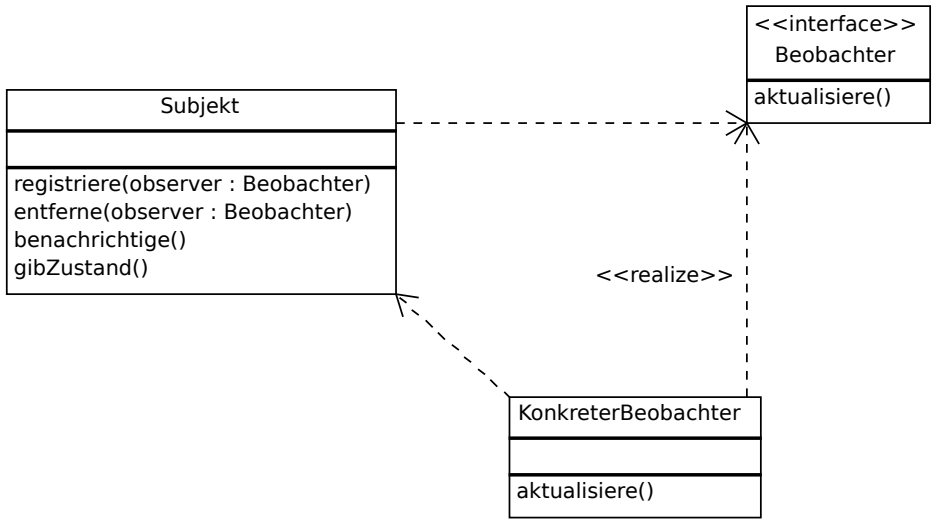


Abbildung 2: Klassendiagramm eines allgemeinen Observer Patterns

2.2 Command

Das *Command*-Design-Pattern ist eine Lösung der *Gang of Four* (GoF) [3] für die Kommunikation zwischen grafischer Oberfläche und Modell/Logik. Eine grafische Oberfläche sendet ein Befehl-Objekt und Modell/Logik können dieses Befehl-Objekt ausführen und archivieren. Befehl-Objekte können nicht nur ausführbar sein, sondern auch beinhalten wie ihre Ausführung rückgängig zu machen ist und somit eine Undo/Redo-Funktionalität bereitstellen [4]. Der Einsatz von Befehl-Objekten erleichtert das Logging von ausgeführten Befehlen, durch die Möglichkeit der Archivierung. Ein Befehl-Objekt kapselt relevante Veränderungen und das Wissen, was der Befehl für eine Veränderung bewirkt.

2.3 Composite

Das *Composite*-Design-Pattern ist auch eine Lösung der GoF [3]. Composite wird verwendet um hierarschische Baumstrukturen aufzubauen. Das Composite-Pattern auf Bäume angewand sieht folgendermaßen aus: Ein Baum besteht aus Knoten. Ein Knoten kennt seine Kindknoten. Diese Kindknoten kennen ihre Kindknoten. Das geht weiter, bis ein Knoten keine Kinder besitzt.
Aufgrund der baumartigen Struktur können Teilbäume durchlaufen werden. Ein Beispiel für die Verwendung des Composite-Patterns ist die im *Java Standard Development Kit* bereitgestellte Swing-Bibliothek. Swing ist eine Bibliothek zum Erstellen von grafischen Oberflächen. Grundlage ist ein JFrame, welcher Components beinhaltet. Components beinhalten wieder Components. Ein JFrame kann in verschiedene Teile unterteilt werden, die Teilbäume darstellen. Abb. 3 verdeutlicht die hierarchische Struktur einer Swing-Oberfläche, die mit dem Composite-Design-Pattern aufgebaut ist.

2.4 Model-View-Controller

Das Design-Pattern *Model-View-Controller* (MVC) [5] ist eine Anwendung verschiedener Design-Patterns um eine Architektur in drei Module zu unterteilen. Das Modell ist hierbei eine Datengrundlage die dem Programm zur Verfügung steht und durch Benutzung des Programms verändert werden kann. Sie ist die unterste Schicht des MVC und ist nur für Datenmanipulation und Datenaufbewahrung zuständig. Die Modellschicht benachrichtigt bei Veränderungen des Modells mittels des Observer-Pattern alle registrierten Listener. Der View ist die Präsentationsschicht und sollte möglichst nur Präsentationskomponenten enthalten und so wenig Funktionalität wie möglich bereitstellen. Der View ist mit dem Composite-Design-Pattern aufgebaut und ist ein Beobachter des Modells. Seine Aufgabe ist neben dem erzeugen der Komponentenstruktur das Aktualisieren der Komponenten bei Zustandsänderungen im Modell. Der Controller kann ebenfalls ein Beobachter des Modells sein und auf die Veränderung reagieren. Im Normalfall ist der Controller jedoch für die Interaktion mit der Benutzeroberfläche zuständig. Das bedeutet der Controller implementiert die Funktionalität der Listener für den View. Ein Controller ist einem View zugeordnet. Somit bilden View und Controller ein Paar. Jeder View kann aber mehreren Controllern zugeordnet sein. Da der Controller für die Interaktion mit dem Benutzer zuständig ist, ist der Controller der Teil, der das Modell verändert.

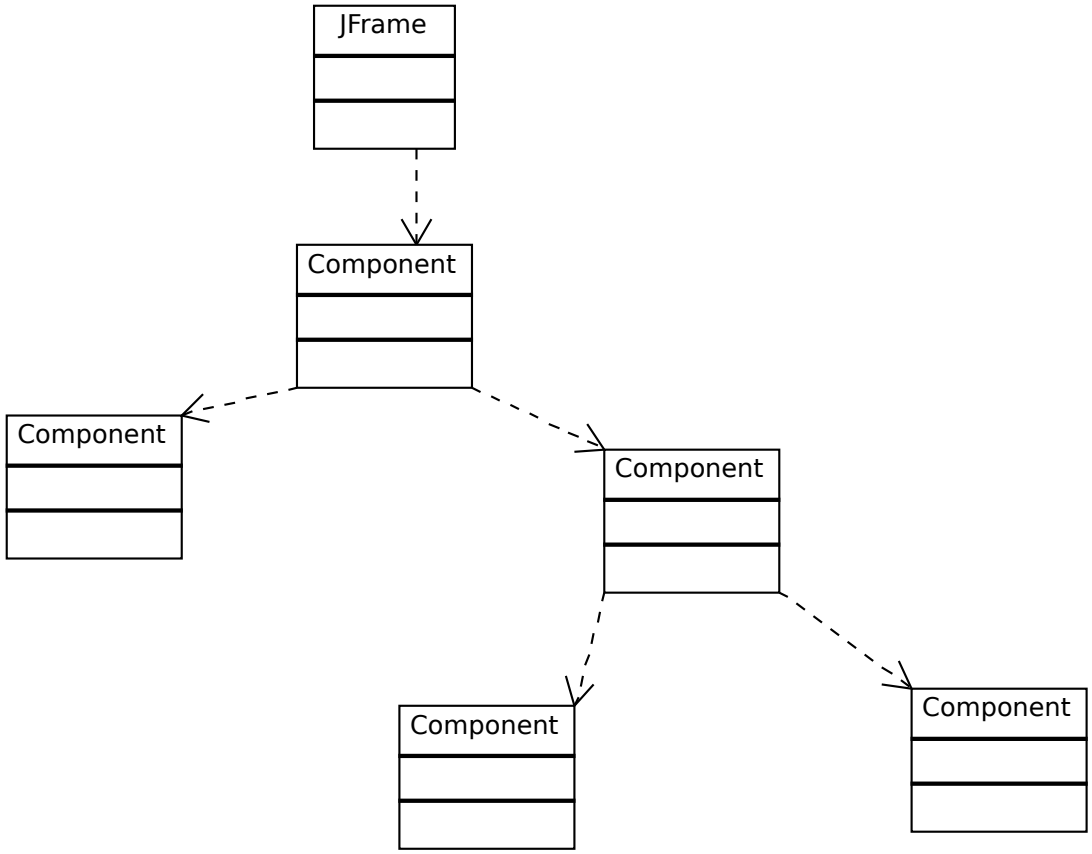


Abbildung 3: Objektdiagramm eines Composites

3 Graphical Editing Framework

Damit bei diesem Projekt nicht von vorne begonnen werden musste, wurden bestehende *Frameworks* verwendet. Frameworks bieten Implementierungen von möglichst allgemein benutzbaren Komponenten, haben jedoch in vielen Fällen eine Struktur, nach der sich der Benutzer des Frameworks richten muss. Für das Projekt ist GEF besonders wichtig, da es die Basis bildet.
GEF ist Teil des *Eclipse Rich-Client-Platform* (Eclipse RCP) Frameworks [6]. Eclipse RCP basiert auf dem OSGi Framework, dass für modulare und skalierbare Software gedacht ist. Eclipse RCP bietet die Möglichkeit auf Basis der in Eclipse RCP bestehenden Komponenten eine eigene Software zu schreiben. Das Framework ist sehr mächtig und besteht aus mehreren Komponenten wie zum Beispiel GEF oder *Standard Widget Toolkit* (SWT) [7], mit dessen Hilfe GEF grafische Komponenten in einem Fenster darstellen kann.
Mit Hilfe von GEF [8] werden grafische Editoren gebaut, wofür es viele benutzbare Features bietet [9]. Zum Erstellen von grafischen Objekten gibt es beispielsweise ein Creation-Tool und das Auswählen von Objekten ist über ein Marquee-Tool bereitgestellt. So lässt sich mit wenig Quelltext ein Kreis auf den Bildschirm bringen, der in der Größe veränderbar ist. GEF ist nach dem MVC-Pattern aufgebaut. Die Ansicht spiegelt ein Modell wieder und wenn das Modell verändert wird, erfährt die Ansicht davon und sorgt dafür,

dass das neue Modell korrekt dargestellt wird. Es werden nicht nur Features geboten, sondern es wird auch direkt eine Architektur mitgeliefert.

4 GEF und die Auswirkung auf die Software Architektur

Da die benutzten Design-Patterns und die aktuelle Software-Architektur betrachtet wurden, ist die Vorarbeit geleistet, um das Zusammenspiel von GEF mit der Software-Architektur zu verstehen. GEF hat eine Struktur bei der für einen Editor eine EditPartFactory erzeugt wird. Die EditPartFactory wiederum erzeugt EditParts. Diese EditParts haben verschieden Funktionen. Zum einen werden die Figures (grafische Abbildung des Modells) erzeugt, zum anderen werden sogenannte EditPolicies installiert, die Möglichkeiten definieren, wie der Benutzer mit dem erzeugten grafischen Element interagieren kann. In den EditPolicies werden die Befehle erzeugt, die mit Hilfe des Command Patterns auf dem Commandstack abgelegt und ausgeführt werden. Ein Command entsteht aus der Interaktion des Benutzers mit dem grafischen Objekt. Zum Beispiel wird in dem Grafikeditor ein Command erzeugt beim Verschieben eines Objektes, welcher die neue Position enthält. Bei Ausführung des Commands wird die neue Position im Modell gespeichert. Das Verändern des Modells bewirkt eine Benachrichtigung der Beobachter. Es wird also erwartet, dass es EditPart, EditPolicies, Figures und Commands gibt. Dadurch ist eine gute Struktur vorgegeben, die dem MVC entspricht. Die vorgegebene Struktur findet sich auch in unserer Paket-Struktur wieder.
Auch die Figures enthalten eine bereits vorgegebene Struktur. Sie sind durch das Composite-Pattern hierarchisch gegliedert. Das bedeutet eine Figure kann beliebig viele Kinder haben, welche wiederum Kinder haben können. Das Prinzip des Aufbaus der Figures ist in Abb. 3 visualisiert.
Es gibt also Teile der Architektur, die vorgegeben ist, allerdings unterliegt jede Architektur im Laufe eines Projektes einem evolutionären Prozess. Die Architektur wächst und wird dabei unstrukturierter. Daher bringt jede Evolution einer Software ein Refactoring mit sich. Auch die Vorgabe einer Struktur kann das nur in seltenen Fällen verhindern.

5 Programme zur Software-Architektur Analyse

Zur Analyse der Software-Architektur werden zwei verschiedene Programme benutzt. Beide Programme sind Eclipse-Plugins und leicht über den Marketplace bei Eclipse-Helios zu installieren.
Das erste Programm ist eDepend und ist Teil des Eclipse-Plugins eUML2 [10]. Mit eDepend können sehr leicht Paket/Klassen-Zyklen analysiert werden, da das Programm baumartig darstellt, welche Zyklen und Subzyklen existieren und alle beteiligten Pakete anzeigt. Allerdings kann das Programm nicht anzeigen durch welche Klassen der Zugriff auf Pakete erfolgt, weswegen beim Beheben jede Klasse einzeln auf Abhängigkeit zu betroffenen Paketen untersucht werden muss. Das Werkzeug kann außerdem eine gesamte Abhängigkeitsdarstellung, sowohl von Klassen, als auch von Paketen, erzeugen. Die Darstellung ist allerdings bei diesem Projekt sehr unübersichtlich und der Nutzen ist gering.

Das zweite Programm ist von Google und heißt Google CodePro Analytix [11]. CodePro Analytix ist umfangreicher als eDepend. Es generiert Metriken nach verschiedenen Kriterien, kann nicht benutzten Quelltext oder Quelltext der gleich zu sein scheint auffinden und kann nach verschiedenen Kriterien den Quelltext nach schlechtem Programmierstil durchsuchen. Dazu werden mehrere Kriterien mitgeliefert, die auf unterschiedliche Sachen achten. Paketabhängigkeiten können visualisiert werden, allerdings ist es in der Hinsicht nicht so komfortabel wie eDepend. Zur Analyse der Software-Architektur haben sich für dieses Projekt besonders die Metriken bewährt, die nützliche Statistiken liefern.

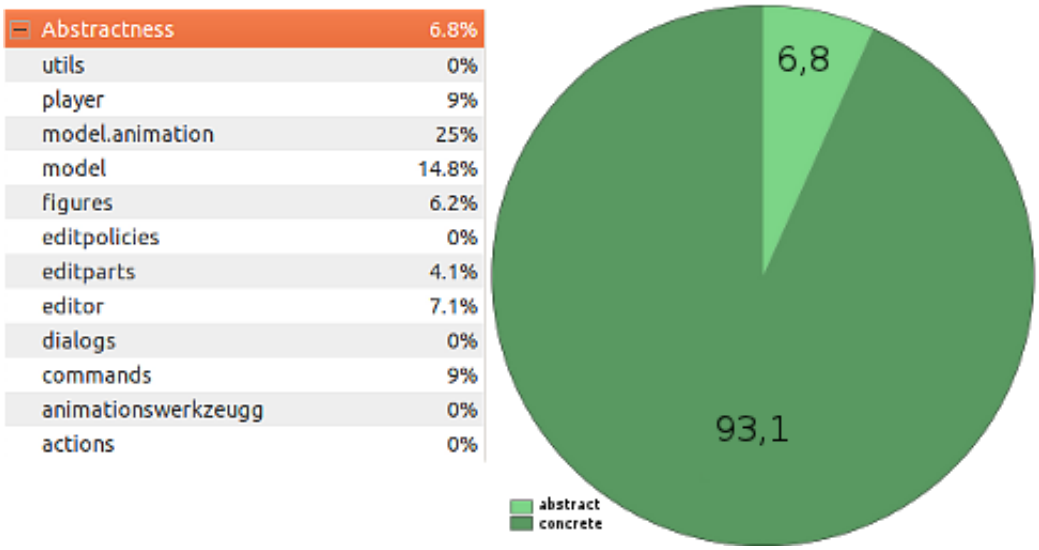
6 Analyse der Software-Architektur

Richtig benutzte Design-Pattern verbessern zwar den Quelltext, garantieren aber noch keine gute Software-Architektur. Einige Design-Pattern wie MVC zielen auf eine Architekturverbesserung ab, schaffen allerdings nur eine grobe Vorgabe, die genug Freiraum lässt, um eine schlechte Architektur im Laufe eines Projektes entstehen zu lassen. Aus diesem Grund müssen die Entwickler aufpassen, dass sie bestimmte Regeln einhalten, um eine Architektur nicht unkontrolliert wachsen zu lassen. Es gibt eine Regel, die besonders herausgegriffen werden soll, da sie ein Grundprinzip der Objektorientieren Programmierung bildet. Die Paket- und Klassenstruktur soll azyklisch sein (Acyclic Dependency Principle) [12]. Ein Zyklus zwischen Paketen/Klassen entsteht, wenn beide Pakete/Klassen voneinander abhängig sind. Das kann über eine direkte Abhängigkeit geschehen oder eine transitive Abhängigkeit. In der direkten Abhängigkeit sind zwei Pakete/Klassen involviert, in der transitiven Abhängigkeit entsteht eine Verkettung von Abhängigkeiten zwischen Paketen oder Klassen, die bei dem Startpaket/der Startklasse endet.
In der Software-Architektur werden zuerst direkte Zyklen zwischen Paketen/Klassen eliminiert. Meistens folgt daraus eine Eliminierung der transitiven Zyklen oder eine Verkleinerung der transitiven Zyklen, die dann leichter zu eliminieren sind. Aus diesem Grund werden in den nächsten Abschnitten direkte Paket- und Klassenzyklen betrachtet und Lösungsvorschläge herausgearbeitet.

6.1 Interface-Paket zur Reduktion von Abhängigkeiten zu konkreten Klassen

Eine in der Literatur oft genannte Lösung zur Eliminierung der zyklischen Paket Abhängigkeiten ist das Einführen eines Interfaces oder das Verschieben von Klassen/Methoden, da sie falsch platziert wurden [12]. Dadurch werden Abhängigkeiten von instabilen Klassen reduziert und die Abhängigkeit zu abstrakten stabilen Klassen verlagert. Insgesamt gibt es sehr viele Abhängigkeiten von instabilen Klassen. Abhängigkeiten von instabilen Klassen sind schlecht, da dann Teile des Programms nicht unabhängig voneinander veröffentlicht werden können. Daher sollte versucht werden, so wenig Abhängigkeiten wie möglich in die Architektur mit einzubringen. Durch Einführen von Interface-Paketen kann eine niedrige Kopplung zwischen instabilen Klassen erreicht werden. Der Vorteil hieran ist, dass Klassen, die nur Schnittstellen benötigen, keine Abhängigkeiten zu Implementationen besitzen. Interface-Pakete haben eine hohe Abstraktionsrate und die Instabilitätsrate ist bei null. Das bedeutet, dass andere die Schnittstellen gut importieren

Abbildung 4: Abstraktionsrate des Projektes vor dem Refactoring. Erzeugt durch Google CodePro Analytix



können, ohne das sie Fehler mit in das Projekt einbringen. In Abb. 4 ist eine Darstellung über die aktuelle Abstraktionsrate zu finden. Die Abstraktionsrate ist ein Verhältnis aus der Summe der abstrakten Klassen und Interfaces zu allen Klassen. In diesem Fall ist die Abstraktionsrate von 6,8% ziemlich gering und kann durch konsequentes Einführen von Interfaces auf über 20% (Status nach dem Refactoring: 20,3%) gehoben werden.

Durch das Benutzen von Interface-Paketen erhöht sich die Wartbarkeit des Projektes, da einzelne Implementationen der Interfaces ausgetauscht werden können. Dadurch ist es möglich fehlerhafte oder schlecht programmierte Implementationen leicht zu erneuern oder anzupassen.

6.2 Paket-Zyklen

Zunächst wird betrachtet, warum Paket-Zyklen so schlecht für die Software-Architektur sind und weshalb sie eliminiert werden sollten, sobald sie entdeckt werden.
Zyklische Abhängigkeiten zwischen Paketen hat folgende vier von Wilcox [13] genannten negativen Auswirkungen:

1. Vermindert die Möglichkeit, Teile der Architektur getrennt von ihr zu benutzen.
2. Pakete in zyklischer Abhängigkeit müssen als eine Einheit veröffentlicht werden.
3. Veränderungen wirken auf Teile der Architektur ein, bei denen die Einwirkung nicht zu vermuten ist.
4. Zyklische Abhängigkeiten zwischen den verschiedenen Architekturebenen bindet die Ebenen und der positive Effekt von Ebenen verschwindet

Bei Betrachtung fällt auf, dass die zyklischen Abhängigkeiten zunächst kein Problem darstellen, da alle negativen Auswirkungen sich auf eine spätere Überarbeitung des Programms beziehen. Große Projekte werden allerdings häufig erweitert, verändert oder es

werden Fehler korrigiert, weshalb so früh wie möglich damit begonnen werden sollte, die Abhängigkeiten aufzulösen.
In dem Projekt AVsH gibt es fünf direkte Zyklen, die zum Teil als Unterzyklen in transitiven Zyklen enthalten sind.

6.2.1 Zyklus zwischen editParts und editPolicies

Die EditParts haben eine Abhängigkeit zu den EditPolicies. Diese Abhängigkeit ist gewollt, da ein EditPart mehrere EditPolicies installieren kann. Die Abhängigkeit von EditPolicies zu EditParts ist jedoch semantisch nicht korrekt. Eine EditPolicy braucht keinen EditPart, da sie nur einen Befehl erzeugen soll, der ein Modell verändert. Mit dem Erzeugen von Befehlen und dem Verändern vom Modell, hat ein EditPart nichts zu tun, da er rein für das Erzeugen von visuellen Komponenten zuständig ist.
Die Abhängigkeit zu dem Paket editParts entsteht durch die Benutzung der Klasse AbstractEditPart in den zwei Klassen ChangeAnimEditorLayout und ChangeGraphEditorLayoutPolicy. Beide Klassen benutzen den AbstractEditPart um eine Typprüfung zu ermöglichen. Diese Typprüfung geschieht auf zwei verschiedenen Wegen. In ChangeGraphEditorLayoutPolicy wird instanceof benutzt, in ChangeAnimEditorLayout wird eine Methode aufgerufen, die erst nach einer Typumwandlung zu AbstractEditPart bereitsteht, anhand deren Rückgabewert der Typ bestimmt wird.
Die problematischen Klassen und die Abhängigkeit ist nun bekannt. Eliminiert werden kann diese durch die Einführung eines Interfaces und Auslagerung des Interfaces in ein Interface-Paket oder aber durch die Auslagerung der abstrakten Klasse AbstractEditPart in ein extra Paket.

In Abb. 5 ist die derzeit im Projekt verwendete Version zu sehen. Da AbstractEditPart Teil des Pakets editParts ist, entsteht die Abhängigkeit. In Abb. 6 ist eine mögliche neue Struktur zu sehen. Das neue Interface OurEditPart enthält eine Methode, die von AbstractEditPart implementiert wird. OurEditPart ist Teil des neu eingeführten Paketes interfaces. Somit sind alle EditPolicies und der AbstractEditPart von interfaces abhängig, untereinander besteht aber keine Abhängikeit mehr.

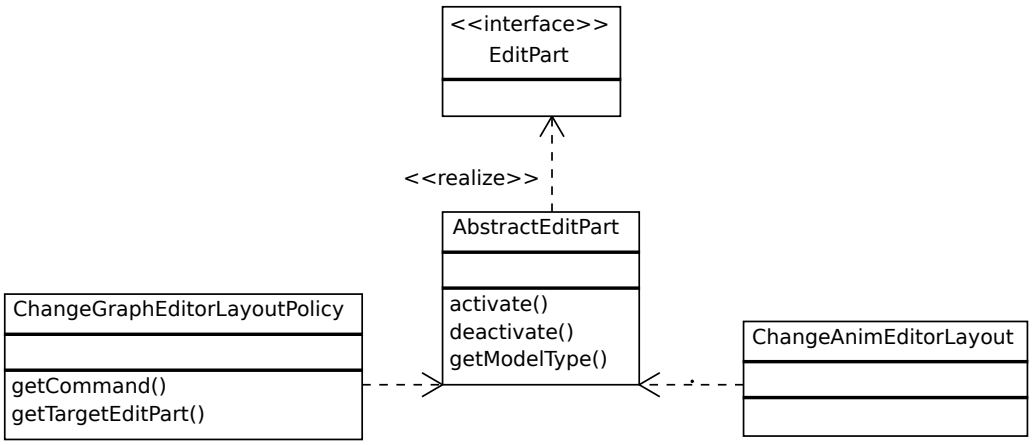


Abbildung 5: Abhaengigkeit derzeit im Projekt

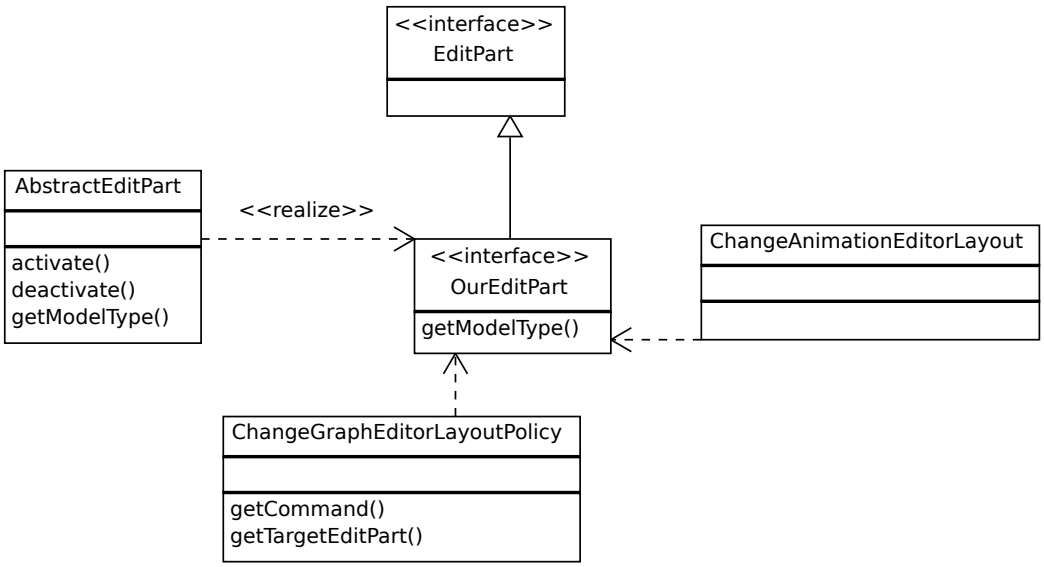


Abbildung 6: Abhängigkeit nach Einführung des Interface

6.2.2 Zyklus zwischen editor und player

Ein weiterer direkter Zyklus besteht zwischen den Paketen editor und player. Nach der in Abb. 1 zu Beginn herausgearbeiteten Grundstruktur soll der Player aus dem Modell das Dokument bekommen. Demzufolge dürfte es keine Abhängigkeit zwischen dem Player und dem Editor geben, sondern nur eine Abhängigkeit zwischen dem Player und dem Modell.

Im Paket player wird die Klasse MultiEditorInput aus dem Paket editor in der Klasse ApplicationState benutzt. Die Klasse ApplicationState ist eine Klasse zur Verwaltung eines SVG-Dokuments und zur Benachrichtigung des Players, wenn ein SVG-Dokument vorhanden ist. Bei erneuter Betrachtung der Grundstruktur stellt dies die Beziehung zwischen Modell und Player her und zeigt, dass die Klasse im falschen Paket ist. Diese Klasse sollte Teil des Modells sein.

Bei Verschiebung der Klasse in das Paket model wird bei der Zyklenbekämpfung zunächst nicht viel gewonnen, da eine transitive Abhängigkeit entsteht zwischen model, editor und player. Doch das Einführen eines Interfaces für die Klasse MultiEditorInput hilft die Abhängigkeit zu eliminieren, wenn das Interface konsequent benutzt wird.

6.2.3 Zyklus zwischen model und model.animation

Der nächste betrachtete Zyklus, besteht zwischen model und model.animation. Vom semantischen macht es Sinn, dass model.animation auf model zugreift. Deshalb soll das Paket model nicht mehr auf das Paket model.animation zugreifen.
Die Verbindung zu model.animation besteht in den zwei Klassen NodeCreationFactory und SceneNodePropertySource. Zunächst wird die Auflösung des Zugriffs in der Klasse SceneNodePropertySource betrachtet. Die Klasse benutzt in fünf Methoden sehr viele if/else Kontrollstrukturen was zu einer Aufblähung führt und ein Gesamtrefactoring nötig macht. Durch die gesamte Neustrukturierung der Klasse wird für die Aufhebung der

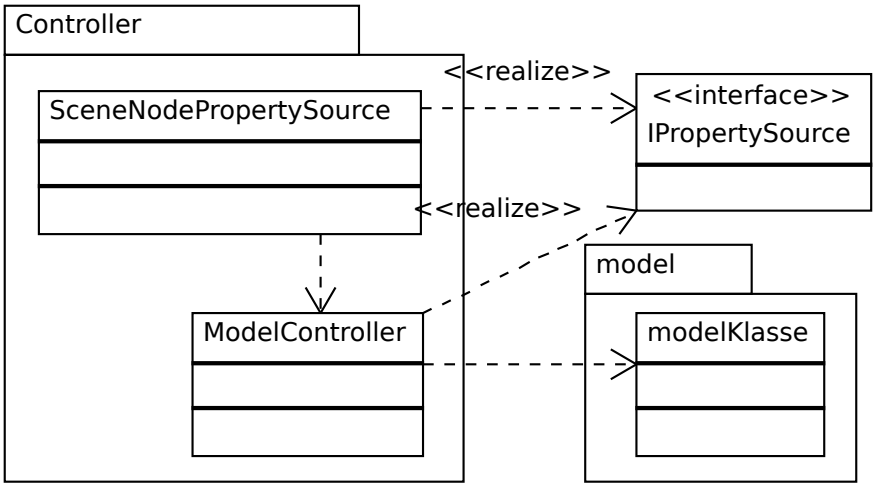


Abbildung 7: Mögliche neue Struktur durch Refactoring der Klasse SceneNodePropertySource

Abhängigkeit zu model.animation gesorgt.

Zunächst sollten zwei neue Interfaces in dem Paket interfaces eingeführt werden um model.animation-Klassen von model-Klassen zu trennen. Die Interfaces stehen in einer Vererbungsbeziehung. IModelAnimation erbt von IModel. IModel besitzt fünf Operationen, die jede model-Klasse implementieren muss, weswegen die abstrakte Klasse SceneNode das Interface implementiert und die Methoden abstract deklariert. Dies sind die Operationen getPropertyDescriptor(), getPropertyValue(Object), isPropertySource(Object), resetPropertyValue(Object), setPropertyValue(Object,Object). Die Implementation der Operationen ist im wesentlichen in der Klasse SceneNodePropertySource vorhanden und muss auf die jeweilige von SceneNode erbende Klasse aufgeteilt werden. Die einzelnen if/else Kontrukte können durch eine einfache Delegation an die Exemplarvariable SceneNode weitergereicht werden.
Achtung: Bei einer derartigen Aufteilung der Methoden, wird Model mit View vermischt, da offensichtlich im Model View-Elemente erzeugt werden, daher wäre es ratsam diesen Teil in den Controller auszulagern. Eine mögliche Anpassung ist, für jede Klasse im Model eine Controller-Klasse zu schreiben, die das Interface IPropertySource implementieren. In Abb. 7 ist die neue Struktur nach dem diskutierten Refactoring dargestellt.
Bisher wurde nur SceneNodePropertySource betrachtet. Um den Zyklus jedoch komplett aufzulösen, muss die Klasse NodeCreationFactory noch überarbeitet werden. Diese Klasse ist im Vergleich zur SceneNodePropertySource ein geringes Problem, da in nur einer Methode mit Hilfe von Fallunterscheidungen Objekte erzeugt werden.

Es bietet sich an, die Objekte mit Hilfe der Reflection-API zu erstellen und so die Abhängigkeit zu anderen Modell-Klassen zu eliminieren. Mit Hilfe von Reflection kann ein Objekt aus dem Namen der Klasse erzeugt werden. Dabei wird die Frage aufgeworfen, ob die Klasse im richtigen Paket ist, da sie nun fast keine Abhängigkeit zu dem Modell mehr hat, semantisch aber sehr eng zu der Klasse MultiEditorInput gehört, daher ist es sinnvoll, beide im gleichen Paket zu haben. Das Benutzen von Reflection ist die einfachste Lösung, allerdings ist es keine schöne Lösung. Reflection hat den Nachteil, dass es lang-

Tabelle 1: Liste von Klassenzyklen und der Vorschlag zum Beheben

Klasse1	Klasse2	Vorschlag
SceneNode	Path	Interface OurNode für SceneNode
MultiEditorInput	GraphicalEditor	Interface OurEditorInput für MultiEditorInput
MultiEditorInput	AnimationEditor	Interface OurEditorInput für MultiEditorInput
MultiEditorInput	SceneNode	Interface OurNode für SceneNode
Animation	AnimationListener	AnimationListener in Interface-Paket
Animation	AnimationPoint	Interface OurAnimationPoint für AnimationPoint
NodeCreationFactory	MultiEditorInput	Interface OurEditorInput für MultiEditorInput

sam ist und schwer nachzuverfolgen, da im Quelltext keine Abhängigkeit mehr auftritt. Nach dem Refactoring und dem konsequenten Einsatz von Interfaces ist die Lösung mit Reflection nicht mehr möglich.

6.2.4 Weitere Paket-Zyklen

Im Projekt befinden sich zwei weitere Paket-Zyklen. Die Behebung läuft im allgemeinen genauso ab, wie die Behebung in den anderen Paket-Zyklen.
Der Zyklus zwischen editor und model wird aufgelöst, indem ein Interface für die Klasse MultiEditorInput bereitgestellt wird. Das entspricht dem MVC-Pattern, da das Modell nichts von dem Editor wissen soll.
Das Paket editor ist in einen weiteren Paket-Zyklus mit dem Paket actions verstrickt. Der Zyklus wird in dem Paket actions behoben, indem für die Klassen AnimationEditor und MultiEditDomain Interfaces eingeführt werden. Die Abhängigkeit von actions zu editor ist nicht sinnvoll, da eine Action zu einem Editor gehört und somit kein Exemplar von einem Editor erzeugen will.

6.3 Klassen-Zyklen

Ebenso wie Paket-Zyklen sind auch Klassen-Zyklen schlecht für die Software-Architektur. Die negativen Aspekte der Klassen-Zyklen sind vergleichbar mit den negativen Aspekten der Paket-Zyklen. Allerdings lassen sich Klassen-Zyklen viel leichter aufheben, da in direkten Zyklen nur zwei Klassen verwickelt sind, während bei Paketen alle Abhängigkeiten zwischen den Paketen untersucht werden müssen, wie bei der Analyse der Pakete model und model.animation zu sehen ist. Klassen-Zyklen treten in diesem Projekt häufiger auf als Paket-Zyklen. Insgesamt sind es sieben direkte Klassen-Zyklen, wobei alle durch konsequentes Benutzen von Interfaces behoben werden können. In Tab. 1 sind die abhängigen Klassen aufgelistet, wobei die letzte Spalte bereits einen Vorschlag gibt, wie der Zyklus behoben werden kann.

6.4 Erweiterbarkeit testen

Ein Test auf Erweiterbarkeit ist nicht ganz einfach, da es keine Programme gibt, die dies übernehmen können. Daher gibt es ein Verfahren um dies zu testen, dass sich *Scenario-*

based Architecture Analysis Method (SAAM) [14] nennt. Bei SAAM werden Szenarien erstellt und mit Hilfe eines Teams die Architektur anhand der Möglichkeit der Durchführung der Szenarien bewertet. Es können für die Qualitätsmerkmale Verfügbarkeit, Zuverlässigkeit, Wartbarkeit, Performanz, Modifizierbarkeit, Erweiterbarkeit, Ökonomie und Portierbarkeit einer Software-Architektur Szenarien erstellt werden. Die Merkmale Verfügbarkeit, Performanz, Ökonomie und Portierbarkeit werden vernachlässigt, da die Merkmale für das Projekt nicht so wichtig sind.
Exemplarisch werden Szenarios zur Erweiterbarkeit und Wartbarkeit durchgegangen, da sie eng aneinander gekoppelt sind. Ein Programm, dass leicht erweiterbar ist, ist meist auch leicht wartbar, weswegen die Erweiterbarkeit im Vordergrund steht.
Vereinfacht werden zwei Szenarien von einer einzelnen Person durchgegangen und das Ergebnis aufgeteilt in zwei Merkmale. Zuerst wird betrachtet, ob es möglich ist, die Software nach den Szenarien zu erweitern ohne bedeutende Architekturveränderungen zu machen. Möglich ist es dann, wenn nach Ablauf einer Zeit eine Lösung gefunden wurde, das Projekt zu erweitern. Wenn es möglich ist, wird die Anzahl der Klassen, die neu erstellt oder bearbeitet werden müssen, betrachtet. Dies gibt einen Überblick über die Modifizierbarkeit und Wartbarkeit.
Die ersten zwei Szenarien testen die Erweiterbarkeit der beiden Editoren um ein typisches Element für den Editor. Beim Grafikeditor wird ein neues Element Trapez hinzugefügt, beim Animationseditor wird, ungeachtet, ob dies in SVG möglich ist, eine neue Animationsart hinzugefügt.
Um zu erkennen, ob die Architektur gut ist, werden die Szenarien zusätzlich für die Architektur nach dem Refactoring durchgegangen.

6.4.1 Szenario 1: Zur Verfügung stellen eines Trapezes im Grafikeditor

In der alten Architektur muss zuerst eine neue Klasse im Modell angelegt werden, die von der Klasse Shapes erbt. In dieser Klasse stehen Informationen des Modells. Zu jedem grafischen Objekt gibt es eine Figure-Klasse, die die grafische Darstellung des Modells im Grafikeditor übernimmt. Um ein im Grafikeditor erzeugtes Trapez auch im Animationseditor korrekt anzuzeigen wird eine neue Figure-Klasse benötigt. Das Erzeugen der Figure übernimmt ein EditPart. Es werden also zwei EditParts gebraucht, die diese Aufgaben übernehmen.

Bisher war das Anlegen eines neuen graphischen Elements kein Problem und es sind derzeit fünf neue Klassen erzeugt worden. Im folgenden Verlauf wird es etwas schwerer. Bisher bestehende Klassen müssen angepasst werden.

Zuerst werden zum Erstellen/Löschen sechs Befehle gebraucht, die an das Modell gesendet werden. Diese sind vorhanden und müssen lediglich um drei Zeilen erweitert werden. Was getan werden muss, ist in diesen Klassen klar, allerdings werden Fragen aufgeworfen. Die Objekte werden, nach einer Fall-Unterscheidung welcher Model-Typ vorliegt, in ihren Modell-Typ gecastet und anschließend in einem Objekt vom Typ SceneNode gespeichert, wodurch der Cast zuvor unnütz ist. Wozu die Fallunterscheidung dient, ist nirgends vermerkt.

Als nächstes müssen die EditPartFactorys zur Erzeugung der von uns erstellten EditParts angepasst werden. Auch das ist in wenigen Zeilen erledigt. Zu guter letzt müssen zwei Klassen im Modell angepasst werden. NodeCreationFactory ist schnell angepasst,

Tabelle 2: Übersicht der Ergebnisse des 1. Szenarios

	Alte Architektur	Neue Architektur
Anzahl neuer Klassen	5	6
Anzahl zu verändernder Klassen	10	4

denn das System Fallunterscheidung-Typecast setzt sich fort. SceneNodePropertySource zu verändern dauert länger und ist fehleranfällig, da die Klasse sehr groß ist und leicht ein Punkt, an dem etwas hinzugefügt werden muss, übersehen wird. Das Prinzip bleibt aber wie bei den anderen Veränderungen.

Nach dem Refactoring muss zunächst ein Interface ITrapez, welches von dem Interface IMultiPointShape im Paket Interfaces erbt, sowie eine Klasse im Modell Trapez, die das Interface implementiert, erzeugt werden. Nun werden wie zuvor die Figures für den Grafikeditor und den Animationseditor erstellt, die die Trapeze im Grafikeditor und im Animationseditor korrekt darstellen. Ebenfalls wie zuvor werden zwei EditParts benötigt um die Figures für Grafikeditor und Animationseditor zu erstellen.

Auch nach dem Refactoring kommt es noch zu dem schwierigeren Teil. Allerdings fallen durch Veränderungen in den Commands das Bearbeiten von diesen sechs Klassen weg. Dies passiert jetzt für jedes Objekt vom Typ OurNode beziehungsweise EdgedModelElement automatisch. In den zwei EditPartFactory Klassen müssen weiterhin unsere EditParts für das Trapez erstellt werden. Bei den Klassen NodeCreationFactory und SceneNodePropertySource hat sich nichts verändert, weshalb auch hier die beiden Klassen angepasst werden müssen.

Fazit des ersten Szenarios: Eine Anpassung durchzugehen ist in beiden Fällen möglich und durch die wenigen Veränderungen in den einzelnen Klassen ist auch der Programmieraufwand entsprechend gering. Insgesamt müssen in der alten Architektur fünf Klassen neu erzeugt werden und zehn weitere verändert werden. Die Veränderungen sind meist nur wenige Zeilen und aus dem vorherigen Vorgehen ersichtlich. Durch die vielen Fallunterscheidungen ist der Quelltext in SceneNodePropertySource schwerer zu bearbeiten. Bei der neuen Architektur müssen sechs neue Klassen erstellt werden, wobei eine ein Interface ist. Allerdings müssen nur noch vier weitere verändert werden.
Allgemein sind Bearbeitungen von Klassen schwerer, als neue Klassen im Kontext zu erstellen. Deswegen gibt es das Prinzip der Objektorientieren Programmierung, dass offen für Erweiterbarkeit, allerdings geschlossen für Veränderungen programmiert werden sollte [5]. Insgesamt muss also in der neuen Architektur ein Interface zusätzlich zu den fünf Klassen in der alten Architektur erstellt werden, allerdings fällt das Bearbeiten von sechs Klassen weg. Eine Übersicht des Ergebnisses ist in Tab. 2 zu sehen.

6.4.2 Szenario 2: Eine neue Animationsart

In der alten Architektur muss zum Erstellen einer neuen Animationsart zunächst ein neues Modell für die Animation erzeugt werden. Dann soll die Animationsart im Animationseditor erzeugt werden, wozu eine Schaltfläche zur Verfügung stehen muss. Dazu brauchen wir eine neue Klasse in dem Paket actions, die ganz ähnlich den anderen 4 Klassen

aufgebaut sein kann, die in dem Paket schon existieren. Damit sie allerdings aufgerufen werden kann, muss sie noch in der Klasse AnimationEditorContext registriert werden und im Animationseditor muss die neue Schaltfläche erzeugt werden. Jetzt existiert eine neue Schaltfläche um eine Animationsart erzeugen zu können.

Je nachdem was es für eine Animationsart ist, wird ein Dialog benötigt, um grundsätzliche Einstellungen machen zu können. Im aktuellen grafischen Editor existieren für 3 von 4 verschiedenen Animationsarten Dialoge. Damit die Informationen im Modell ankommen, muss allerdings noch die Klasse CreateAnimationPolicy verändert werden und ein entsprechender Command in der Klasse AnimationCreateCommand erzeugt werden. Im Modell muss durch die Klasse NodeCreationFactory ein Objekt der neuen Animationsart erstellt werden und die Klasse SceneNodePropertySource muss abgeändert werden, damit ein PropertySheet für diese Animation bereit steht.

Es gibt zwei Stolperfallen, die Schwierigkeiten bereiten können. Eine typische Vorgehensweise eine ähnliche Funktion wie eine schon bestehende einzubringen, ist das Verfolgen von Referenzen der bestehenden Klassen. In der Klasse GraphicsEditPartFactory wird eine Fallunterscheidung auf MotionAnimation getroffen. Diese Fallunterscheidung ist verwirrend, da sie unnötig ist. Beim Einpflegen der neuen Animation kann es passieren, dass hier die Verzweigung erweitert wird und diese mit jeder Erweiterung wächst. Die andere Stolperfalle ist die Klasse AnimationEditorContext. Die Klasse ist nötig, um das ContextMenu einer Animation anzuzeigen und muss abgeändert werden, allerdings besteht keine Abhängigkeit zu den Animationen, so dass das Auffinden ziemlich schwer ist.

In der neuen Architektur verhält sich das Erstellen der Animation genauso wie in der alten Architektur, mit der Ausnahme, dass zusätzlich ein Interface für die Animation erzeugt werden muss. In der Klasse GraphicsEditPartFactory wurde die Stolperfalle entfernt.

Beide Architekturen sind gleich einfach zu überarbeiten und es macht keinen Unterschied zum jetzigen Zeitpunkt mit welcher Architektur gearbeitet wird. Ein Vergleich des Arbeitsaufwandes ist in Tab. 3 zu sehen und es fällt auf, dass es Klassen gibt, wo zum Zeitpunkt des Programmierens unklar ist, ob dort etwas geändert werden muss oder nicht. Die Unklarheiten, ob Klassen neu erstellt oder verändert werden müssen, entstehen aus unterschiedlichen Gründen. Zum einen hängt es von der Art der Animation ab. Zum Beispiel benötigt eine Bewegungsanimation keinen Dialog, die anderen drei bisher bestehenden benötigen allerdings einen. Zum anderen ist es teilweise undurchsichtig ob etwas verändert werden muss oder aus welchem Grund eine Klasse verändert werden muss. In einigen Klassen wird die Bewegungsanimation extra behandelt, ohne Kommentar warum dies geschieht.

7 Verbesserungsvorschläge für die Software-Architektur

Bei der Betrachtung der Software-Architektur wurden bereits Verbesserungsvorschläge eingestreut. Im Allgemeinen lohnt es sich mit dem Anlegen eines Interface-Paketes an-

Tabelle 3: Übersicht der Ergebnisse des 2. Szenarios

	Alte Architektur	Neue Architektur
Anzahl neuer Klassen	2	3
Anzahl zu verändernder Klassen	6	6
Anzahl eventueller Veränderungen	2	2
Anzahl eventueller neuer Klassen	1	1
Stolperfallen	2	1

zufangen und so die Abhängigkeiten zwischen den Klassen zu reduzieren. Dabei werden die ein oder anderen Paket-Zyklen womöglich schon verschwinden und die Arbeit bei der Eliminierung der Paket-Zyklen hat sich reduziert. In der Regel ist das Anlegen von Interfaces in der Software-Architektur eine frühe Phase, daher ist es empfehlenswert bei der weiteren Planung nach dem Refactoring auch hier weiter zu machen, um nicht danach ein erneutes großes Refactoring zu machen.
Sobald es Interface-Pakete gibt, können die Paket- und Klassen-Zyklen nach den oben besprochenen Methoden eliminiert werden. Bei Zyklen ist darauf zu achten, dass es verschiedene Ursachen gibt. Während bei manchen Zyklen unsauber programmiert wurde, kann ein Zyklus auch aus anderen Gründen, die aus der Planung stammen, entstanden sein. Bei den Paketen model und model.animation ist es der Fall, dass dort Klassen semantisch nicht in die Pakete gehören, wodurch große Probleme beim Refactoring entstehen, da solche Klassen häufig zu verschiedenen Paketen gehören können und auf die Semantik eines Paketes abgeändert werden müssen.
Die Klassen SceneNodePropertySource und NodeCreationFactory haben eine starke Kopplung zu dem Modell-Paket. Die starke Kopplung lässt es nicht zu, dass die Klassen einzeln weiter benutzt werden. Die Klasse NodeCreationFactory ist, obwohl der Aufbau unschön ist, relativ leicht erweiterbar, während die Klasse SceneNodePropertySource sehr unübersichtlich und viel zu lang ist. Insgesamt umfasst das Projekt vor dem Refactoring 16885 Zeilen Quelltext. Die leeren Zeilen sind hier mit eingerechnet. Die Klasse SceneNodePropertySource ist 1164 Zeilen lang. Das bedeutet, dass knapp 7% des gesamten Projektes in dieser einen Klasse zu finden ist. Im Bestfall liegt eine gleichmäßige Verteilung vor, daher sollte die Klasse SceneNodePropertySource übearbeitet werden.

8 Fazit

Die Analyse der Software-Architektur ergibt, dass die Architektur des Projekt an vielen Stellen Mängel aufweist. Es besteht eine stabile Grundlage durch die von GEF gegebene Paket-Struktur, allerdings ist an den Abhängigkeiten der Pakete einiges zu verbessern um ein einfaches Weiterentwickeln zu ermöglichen. Mit Hilfe von eDepend und den erarbeiteten Lösungen zur Auflösung der Zyklen können die Abhängigkeiten reduziert werden. Der Quelltext ist einigermaßen sauber, allerdings lässt sich durch das Werkzeug „Audit Code“ des Google CodePro Analytix Programms der Quelltext auf einen Standard bringen, um späteres Einarbeiten zu erleichtern.
Anhand des ersten Szenarios ist erkennbar, wie stark die Auswirkungen eines Refactorings sein können. Das zweite Szenario hingegen zeigt, dass die Architektur durch ein

Refactoring zwar verbessert wird, allerdings der Arbeitsaufwand nicht für jede Erweiterung oder Wartung vereinfacht wird. Außerdem deckt das zweite Szenario Schwierigkeiten beim Erweitern durch unvollständige Dokumentation auf. Die Metriken von Google CodePro Analytix enthalten eine Statistik über die Dokumentation des Quelltextes. Die Statistik gibt die Zeilen der Dokumentationen pro Zeile Quelltext an und ist in dem Projekt bei 16,2%. Das zeigt mangelnde Dokumentation.

Abkürzungsverzeichnis

GEF Graphical Editing Framework

SVG Scalable Vector Graphics

AVsH Animationswerkzeug zur Visualisierung sozialen Handelns

GoF Gang of Four

MVC Model-View-Controller

Eclipse RCP Eclipse Rich-Client-Platform

SAAM Scenario-based Architecture Analysis Method

SWT Standard Widget Toolkit

Abbildungsverzeichnis

Literatur

[1] Fritz Heider and Marianne Simmel. An experimental study of apparent behavior. *The American Journal of Psychology*, 57(2):243–259, April 1944.

[2] Eric Freeman, Elisabeth Freeman, Kathy Sierra, and Bates Bert. *Entwurfsmuster von Kopf bis Fuß*. O'REILLY, 2006.

[3] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.

[4] Chris Lasater. Design patterns: Command pattern. http://www.codeproject.com/KB/books/DesignPatterns.aspx, 2007. zuletzt abgerufen am 20.03.2011.

[5] Bernhard Lahres and Gregor Rayman. *Praxisbuch Objektorientierung*. O'REILLY, 2006.

[6] Eclipse Foundation. Eclipse rcp. http://wiki.eclipse.org/index.php/Rich_Client_Platform, 2011. zuletzt abgerufen am 20.03.2011.

[7] Eclipse Foundation. Swt. http://www.eclipse.org/swt/, 2010. zuletzt abgerufen am 20.03.2011.

[8] Anthony Hunter. Eclipse rcp - graphical editing framework. http://www.eclipse.org/gef/, 2011. zuletzt abgerufen am 18.03.2011.

[9] Bill Moore, David Dean, Anna Gerber, Gunnar Wagenknecht, and Phillipe Vanderheyden. Eclipse development using the graphical editing framework and the eclipse modeling framework. http://publib-b.boulder.ibm.com/Redbooks.nsf/RedpieceAbstracts/sg246302.html?Open, 2004. zuletzt abgerufen am 18.03.2011.

[10] Soyatec. euml2. http://www.soyatec.com/euml2/, 2011. zuletzt abgerufen am 20.03.2011.

[11] Google. Google codepro analytix. http://code.google.com/intl/de-DE/javadevtools/codepro/doc/index.html, 2010. zuletzt abgerufen am 20.03.2011.

[12] Kirk Knoernschild. *Java Design: Objects, UML, and Process*. Pearson Education, 2001.

[13] Glen Wilcox. Managing your dependencies with jdepend. http://onjava.com/pub/a/onjava/2004/01/21/jdepend.html, 2004. zuletzt abgerufen am 18.03.2011.

[14] Ralf Reussner and Wilhelm Hasselbring. *Handbuch der Software-Architektur*. dpunkt.verlag, 2006.