Diplomarbeit

Eine Plugin-Architektur für Renew

Konzepte, Methoden, Umsetzung

Jörn Schumacher

Betreuung
Dr. Daniel Moldt
Prof. Dr. Winfried Lamersdorf

Universität Hamburg, Fachbereich Informatik

Inhaltsverzeichnis

Kapitel 1

Einleitung

Der am Arbeitsbereich TGI des Fachbereichs Informatik an der Universität Hamburg entwickelte Petrinetzsimulator *Renew* [KW03] hat sich als wertvolles Werkzeug zur Modellierung komplexer Systeme erwiesen. Besonders im Zusammenhang mit den MULti-Agenten-Netzen Mulan [Röl99], bei denen mit Hilfe von Referenznetzen ein Rahmenwerk zur agentenorientierten Softwareentwicklung spezifiziert wurde, wandelt sich die Sicht auf *Renew* hin zu einer Verwendung als Agentenplattform und Entwicklungsumgebung. Diese Anwendung in Verbindung mit Mulan eröffnet viele interessante Forschungsgebiete (etwa [Ree04] oder [Car04]).

Während der Arbeit an agentenorientierten Projekten ergeben sich immer wieder Anforderungen, deren Umsetzung in *Renew*-Erweiterungen den Umgang mit Mulan erleichtert. Dabei ergeben sich drei Probleme.

Zum einen verwenden nicht alle Anwender *Renew* zusammen mit Mulan; wird es zu sehr als Agentenplattform ausgerichtet, besteht die Gefahr, dass dies die ursprünglichen Funktionen als Petrinetzsimulator überdeckt. Die Lösung dieses Punktes liegt darin, *Renew* konfigurierbar zu machen, es also dem Benutzer zu ermöglichen, das System nach seinen Bedürfnissen einzurichten.

Zweitens handelt es sich bei *Renew* um ein sehr komplexes Softwaresystem. Je umfangreicher es wird, desto mehr Erfahrung benötigen die Entwickler, um neue Erweiterungen einbauen zu können. Da viele der Programmierer jedoch häufig Teilnehmer studentischer Projekte sind, muss diese Erfahrung zuerst durch langwierige Einarbeitung gewonnen werden. Diese Schwierigkeiten lassen sich durch eine erhöhte Übersichtlichkeit des Codes und Erweiterbarkeit des Systems beseitigen. Zusätzlich kann ein klarer gegliederter Sourcecode besser verwaltet werden.

Drittens ist es generell nicht einfach, *Renew* als Petrinetzsimulator zu erweitern. Im gegenwärtigen Zustand geschieht dies dadurch, dass für einen neuen Petrinetz-Formalismus ein so genannter *Renew-Mode* programmiert wird, der dafür zuständig ist, einen Netzcompiler bereit zu stellen. Außerdem kann er in der grafischen Benutzungsoberfläche spezifische Erweiterungen einbringen. Solche *Mode*s existieren etwa für *Feature Structure*-Netze [Wie01] und *Workflow*-Netze [Jac02]. Es besteht jedoch der Wunsch, diesen Erweiterungs-Mechanismus zu flexibilisieren und zu vereinfachen.

Das Ziel der vorliegenden Arbeit ist es also, *Renew* mit den genannten Eigenschaften Konfigurierbarkeit, Übersichtlichkeit und Erweiterbarkeit auszustatten.

Für die ersten beiden Punkte ist der Bereich der Komponentenorientierung (siehe [Szy02], [Gri98]) geeignet. Im ersten Teil dieser Arbeit erfolgt eine Einführung in die Begriffe dieses Konzeptes sowie die Vorstellung zweier existierender Komponentensysteme.

In der anschließenden Diskussion zeigt sich, dass Komponenten sowohl für eine bessere Konfigurierbarkeit als auch für übersichtlicher strukturierte Software sorgen.

Lediglich in Bezug auf Erweiterbarkeit erfüllen sie, wie sie in dieser Arbeit verstanden werden, nicht die gestellten Erwartungen. Als Lösung wird der Begriff des *Plugins* eingeführt, definiert und existierende Pluginsysteme vorgestellt.

Anschließend wird ein an die Aufgabenstellung angepasstes Modell für ein Pluginsystem erstellt. Dabei fließen die in den vorigen zwei Kapiteln gemachten Erkenntnisse aus der behandelten Grundlagenliteratur sowie aus den analysierten existierenden Systemen ein.

Damit endet der erste, grundlagenorientierte Teil. Der zweite Teil beschreibt die Umsetzung der Konzepte und des vorgestellten Modells in *Renew*. Dies geschieht in drei Schritten.

Als erstes wird ein System entwickelt, das für das Laden und Verwalten von Komponenten geeignet ist. Dies geschieht generisch, die Funktionalität der geladenen Komponenten ist also nicht fest gelegt.

Als zweiter Schritt erfolgt eine Aufteilung des bestehenden *Renew*-Systems in Komponenten. Es wird zunächst anhand verschiedener Kriterien diskutiert, welche Komponenten aus dem vorhandenen Code zu bilden sind. Dabei müssen zyklische Abhängigkeiten zwischen den entstehenden Systemteilen vermieden bzw. beseitigt werden.

Der letzte Schritt zur Einführung von Plugins in *Renew* besteht im Entwurf der Schnittstellen, durch die die entstandenen Komponenten erweiterbar gemacht werden. Dabei werden allgemein einsetzbare Strukturen aus dem Bereich der Entwurfsmuster angewendet.

Abschließend werden die in der Arbeit zusammengetragenen Ergebnisse zusammengefasst und bewertet. Zusätzlich werden weitere Möglichkeiten genannt, die der Umbau *Renew*s in ein pluginfähiges System bietet.

Teil I

Techniken und Methoden

Kapitel 2

Modellierungstechniken

In diesem Kapitel werden die Beschreibungs- und Modellierungstechniken erläutert, die in der vorliegenden Arbeit verwendet werden, um Konzepte zu beschreiben oder deren Umsetzung darzustellen.

Dabei handelt es sich zum einen um die bekannte Unified Modelling Language UML [Obj03c], die Diagramme zur Verfügung stellt, mit denen objektorientierte Systeme beschrieben werden. Damit können sowohl deren statische Struktur als auch ihr Verhalten spezifiziert werden. Die drei in dieser Arbeit verwendeten Diagrammarten, das Klassen-, Sequenz- und das Zustandsdiagramm, werden im ersten Abschnitt erläutert.

Zum anderen ist es auch möglich, die in *Renew* propagierten Referenznetze als Modellierungssprache zu verwenden. Diese haben mehrere Vorteile. So ist *Renew* nicht nur ein Zeichenwerkzeug für diese Netze, sondern kann sie auch als Simulator ausführen. Dadurch ist es möglich, das erzeugte Modell sofort in seinem Ablauf zu beobachten und so eventuelle Fehler früh zu identifizieren. Außerdem sind Petrinetze gut dafür geeignet, den Daten- und Kontrollfluss übersichtlich darzustellen, da unterschiedliche Zeichenelemente für Daten und Aktionen verwendet werden.

Der zweite Abschnitt dieses Kapitels erläutert, wie Referenznetze als Modellierungssprache eingesetzt werden. Dabei werden Grundkenntnisse im Bereich der Petrinetze vorausgesetzt.

2.1 UML

Bei der Unified Modelling Language UML handelt es sich um einen von der Object Management Group OMG [Obj03c] definierten Standard [Obj03b]. Sie besteht aus einem Satz von Diagrammtypen, mit denen sich objektorientierte Systeme beschreiben lassen.

Es würde den Rahmen dieser Arbeit sprengen, eine vollständige Einführung in alle Elemente der UML zu geben. Daher sollen lediglich die hier verwendeten Diagrammtypen erläutert werden. Als weiter gehende Literatur sei etwa [Oes98] empfohlen.

2.1.1 Klassendiagramme

Klassendiagramme sind für die Darstellung der statischen Struktur eines Systems geeignet. Die enthaltenen Elemente sind Rechtecke, die Klassen repräsentieren. Zum anderen gibt es verschiedene Verbindungsarten, mit denen Beziehungen zwischen Klassen dargestellt werden.

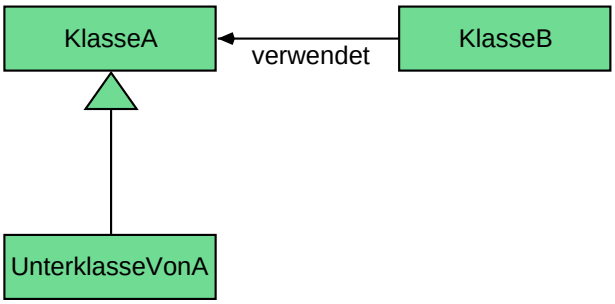


Abbildung 2.1: Beispiel für ein Klassendiagramm

Klassenelemente Abbildung 2.1 zeigt ein Beispiel für ein Klassendiagramm. Daran ist zu sehen, dass die Kästen jeweils mit dem Namen der Klasse beschriftet sind, für die sie stehen. Ebenfalls in UML vorgesehen ist die Möglichkeit, hier auch Methoden anzugeben. Weil Klassendiagramme in dieser Arbeit jedoch lediglich das Verhältnis zwischen Klassen darstellen sollen und ihre Schnittstelle in diesem Zusammenhang nicht von Bedeutung ist, wird davon kein Gebrauch gemacht.

Verbindungselemente In Abb. 2.1 sind zwei Arten von möglichen Verbindungen gezeigt. Der einfache Pfeil zwischen `KlasseA` und `KlasseB` bedeutet eine Assoziation zwischen den beiden Klassen. Der Pfeil ist mit der Art der Beziehung beschriftet: das Beispiel liest sich als *KlasseB verwendet KlasseA*.

Die andere dargestellte Verbindungsart repräsentiert die Vererbungsbeziehung. Für diese wird ein Pfeil mit einer dickeren Spitze verwendet. Das Beispiel zeigt dies zwischen `KlasseA` und `UnterklasseVonA`.

2.1.2 Sequenzdiagramme

Im Gegensatz zum Klassendiagramm ist das Sequenzdiagramm für die Darstellung von dynamischem Systemverhalten gedacht. Dabei wird eine exemplarische Sequenz von Methodenaufrufen zwischen zwei oder mehreren an einem Ablauf beteiligten Klassen dargestellt.

Zu beachten ist, dass ein Sequenzdiagramm lediglich einen idealisierten Fall betrachtet. Es eignet sich, um die Interaktionen zwischen den Exemplaren der dargestellten Klassen zu veranschaulichen. Die Semantik der Aufrufe wird ebenso wenig ausgedrückt wie das Verhalten des Systems im Fehlerfall.

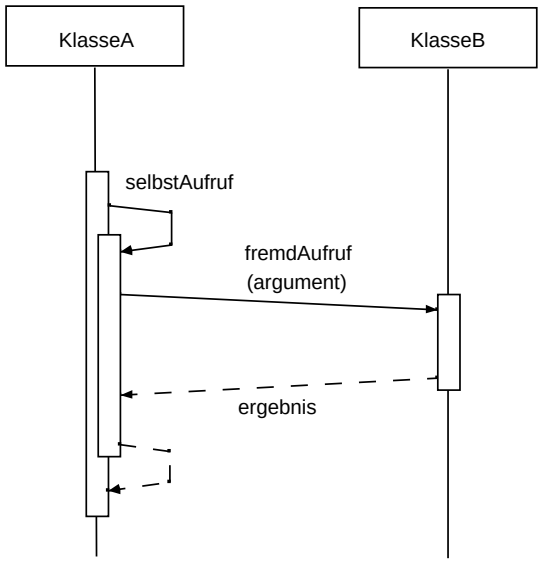


Abbildung 2.2: Beispiel für ein Sequenzdiagramm

Die Elemente eines Sequenzdiagramms sind wie im Klassendiagramm Rechtecke, die Exemplare von Klassen repräsentieren. Diese befinden sich am oberen Rand des Diagramms. Senkrecht darunter befindet sich die Lebenslinie, die die Existenz eines Exemplars der Klasse kennzeichnet. Ist ein Exemplar aktiv, wird dies durch einen Balken auf der Linie dargestellt. Weiterhin gibt es Pfeile, die Aufrufe zwischen den Exemplaren der Klassen darstellen.

Abbildung 2.2 zeigt ein Beispiel für ein Sequenzdiagramm. Die Linien unterhalb der Klassenkästen sind Lebenslinien, die zu einem Kasten werden wenn ein Exemplar der entsprechenden Klasse aktiv ist.

An den Pfeilen zwischen den Klassen sind die Namen der Methoden, die aufgerufen werden, vermerkt, etwaige Argumente stehen in Klammern. Wenn eine Methode verlassen wird und die Kontrolle an den Aufrufenden zurückkehrt, wird dies durch einen gestrichelten Pfeil gekennzeichnet. Dieser kann mit Ergebnissen beschriftet sein.

Im Beispiel ruft ein Exemplar der Klasse `KlasseA` zuerst die eigene Methode `selbstAufruf` auf, in der dann `fremdAufruf` mit dem Argument `argument` auf einem Exemplar der Klasse `KlasseB` ausgeführt wird. Diese liefert das Ergebnis `ergebnis` zurück an `KlasseA`.

2.1.3 Zustandsdiagramm

Zustandsdiagramme dienen ebenfalls der Darstellung dynamischer Eigenschaften. Sie beschreiben jedoch im Gegensatz zu Sequenzdiagrammen nicht die Interaktion zwischen Klassen, sondern die möglichen Zustände eines Exemplars einer einzelnen Klasse. Dabei wird nicht lediglich ein Fall gezeigt, sondern das Gesamtverhalten der Objekte einer Klasse spezifiziert.

Grundelemente eines Zustandsdiagramms sind die Zustände, in denen sich eine Klasse befinden kann. Zustände werden durch Rechtecke dargestellt. Für Anfangs- und Endzustände gibt es spezielle Elemente: einen ausgefüllten Kreis respektive einen ausgefüllten Kreis mit einem Rand. Ein Zustand kann mehrere Unterzustände haben.

Die Zustände sind durch Pfeile miteinander verbunden, die die Zustandsübergänge markieren. Diese sind mit den Namen der Methoden versehen, die den entsprechenden Zustandswechsel auslösen.

In Abbildung 2.3 ist ein Beispiel für ein Zustandsdiagramm zu sehen. Dabei repräsentiert der Kreis links oben den Anfangszustand der Klasse. Durch den Aufruf der Methode `aufruf1` wird der Zustand `äußerer Zustand` betreten, der Unterzustände enthält. Da der Pfeil `aufruf1` nicht direkt in einen dieser Unterzustände führt, aktiviert er den in `äußerer Zustand` enthaltenen Anfangszustand. Analog gilt für den Aufruf von `shutdown`, dass bei Betreten des Endzustandes in `äußerer Zustand` dieser verlassen wird.

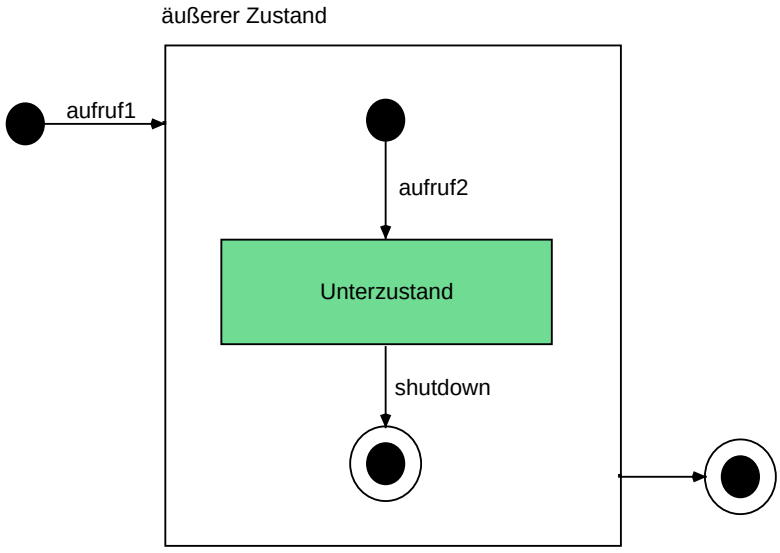


Abbildung 2.3: Beispiel für ein Zustandsdiagramm

2.1.4 Werkzeuge

Durch die weite Verbreitung der UML stehen eine große Anzahl von Werkzeugen zur Verfügung, die ihre Verwendung unterstützen. Die folgende Liste kann daher nur einen ersten Überblick geben und stellt keinen Anspruch auf Vollständigkeit. Viele der Produkte ermöglichen nicht nur den Entwurf mit der UML, sondern unterstützen auch andere Schritte des Entwicklungsprozesses wie etwa automatisches Testen und Konfigurationsmanagement. Häufig können sie auch aus den erstellten Modellen Sourcecode erzeugen.

Rational Rose Bei *Rational Rose* [IBM03b] handelt es sich um eines der ersten und bekanntesten UML-Werkzeuge, von dem inzwischen viele verschiedene Versionen erhältlich sind. Diese unterscheiden sich stark in ihrem Leistungsumfang.

Together *Together* [Bor03] ist wie Rational Rose ein kommerzielles Produkt mit einer Vielzahl an auslieferbaren Versionen.

MagicDraw Auch *MagicDraw* [No 03] stellt in neueren Versionen, wie die beiden vorher genannten Werkzeuge, mehr als nur UML-Modellierung bereit.

Poseidon Das UML-Werkzeug *Poseidon* [gA03] bietet gegenüber dem OpenSource-Werkzeug *Argo/UML* [Arg03], aus dem es entstanden ist, neue Funktionen wie etwa zwei zusätzliche UML-Diagrammtypen und verbesserte Codegenerierung. Die *Community Edition* von Poseidon ist frei erhältlich. Die kommerzielle *Standard Edition* ist – besonders im Rahmen dieser Arbeit – daher interessant, da sie den in [Eic02] beschriebenen Plugin-Mechanismus enthält.

Argo/UML Das letzte Werkzeug in dieser Liste, *Argo/UML* [Arg03], ist ein frei erhältliches Open-Source-Produkt. Die Besonderheit an diesem Produkt ist, dass es den Anwender im Entwurfsprozess unterstützt, indem es Fehler erkennt und sie mit ihrer allgemeinen Lösung in einem eigenen Bereich des Hauptfensters anzeigt. Diese Funktion steht auch in Poseidon (s.o.) zur Verfügung.

2.2 Referenznetze

2.2.1 Bestandteile

In seiner ursprünglichen Form handelt es sich bei *Renew* um einen Simulator für Referenznetze. In diesem Abschnitt erfolgt zunächst eine Einführung in die Elemente von Referenznetzen, die sie von den zu Grunde liegenden einfachen Petrinetzen unterscheiden. Für das Verständnis ist es sinnvoll, mit diesen einfachen Petrinetzen vertraut zu sein, die mit einigen Erweiterungen wie etwa den gefärbten Petrinetzen, beispielsweise in [VG03] erläutert werden. Für eine genaue Spezifikation von Referenznetzen siehe [Kum02]. Referenznetze bieten den Vorteil, dass der Entwickler durch Simulation des Modells mit *Renew* bereits eine Fehleranalyse betreiben kann. Renew selbst ist auf der Homepage [KW03] frei erhältlich.

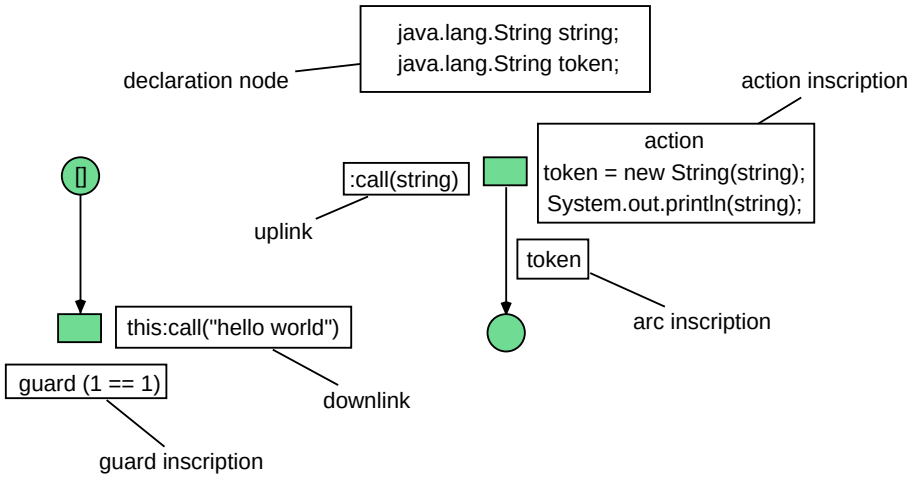


Abbildung 2.4: Beispiel-Referenznetz

Abbildung 2.4 zeigt ein Beispiel für ein Referenznetz mit den dafür spezifischen Elementen. Diese sind durch Boxen hervorgehoben und mit einer Bezeichnung versehen:

1. Der *declaration node* enthält die Deklarationen der Variablen, die in diesem Netz verwendet werden.

2. Eine *guard inscription* an einer Transition ist mit einem Booleschen Ausdruck parametrisiert. Sie verhindert ein Schalten der entsprechenden Transition, wenn der evaluierte Ausdruck nicht *true* ergibt. Der Ausdruck darf beliebig komplex sein. In dem Beispiel auf Abbildung 2.4 ist ein Schalten immer möglich.

3. Ein *uplink* und ein *downlink* bilden einen so genannten *synchronen Kanal*. Die Semantik ist dergestalt, dass die Transitionen mit dem *uplink* und dem entsprechenden *downlink* nur paarweise aktiviert sein und schalten können. Das *this* am *downlink* des Beispiels bedeutet, dass ein *uplink* im gleichen Netz angesprochen werden soll. Grundsätzlich kann es sich hier auch um ein anderes Netz handeln. In den Klammern des synchronen Kanals stehen Variablen, über die Daten zwischen den beiden Transitionen ausgetauscht werden können. Dieser Austausch ist bidirektional möglich.

4. Die *action inscription* an der Transition bewirkt, dass beim Schalten dieser Transition der in der Anschrift enthaltene Programmcode aufgerufen wird. In diesem Beispiel wird der String, der dem *uplink* übergeben wurde (*"hello world"*), ausgegeben und der Variable *token* zugewiesen.

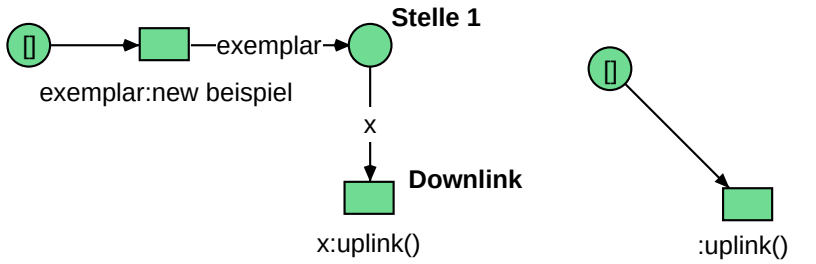
5. Bei *arc inscriptions* handelt es sich um Variablennamen, die ein Token bezeichnen, das aus einer Stelle an eine Transition übergeben wird oder - wie im Beispiel - aus einer Transition in eine Stelle.

Es ist anzumerken, dass es sich bei den Beschriftungen im Beispiel immer um Ausdrücke der Sprache Java handelt, die zur Laufzeit (d.h. während der Simulation) ausgewertet werden. Der Referenznetzformalismus selbst gibt jedoch nicht vor, welche Sprache an den Anschriften stehen kann: Vor der Simulation formt ein Compiler die vorhandenen Netze in ein Format um, das der Simulator dann ausführen kann.

2.2.2 Netze und Netzexemplare

Eine Eigenschaft von Referenznetzen besteht darin, dass es möglich ist, während der Simulation Exemplare der erstellten Netze zu erzeugen. Diese können auf einer Stelle in einem übergeordneten Netz abgelegt werden.

Die Abbildung 2.5 (a) zeigt, wie ein Exemplar eines Netzes mit Namen `beispiel`, wie es auf Abb. 2.5 (b) zu sehen ist, erzeugt und auf der Stelle `Stelle 1` abgelegt wird. Die daran angeschlossene Transition `Downlink` zeigt, wie sie die beiden Netze durch einen synchronen Kanal synchronisieren.



(a) Erzeugendes Netz

(b) In (a) erzeugtes Netz `beispiel`

Abbildung 2.5: Erzeugung eines Netzexemplars

2.2.3 Modellierung mit Referenznetzen

Durch die enge Bindung von Referenznetzen an Java ist es möglich, sehr komplexe Systeme mit Hilfe von *Renew* zu entwerfen. Ein Beispiel für ein solches System ist etwa das Multiagentensystem MULAN [Röl99], das fast ausschließlich durch Referenznetze modelliert wurde. Dabei kann der Simulator von *Renew* dafür verwendet werden, das erstellte Modell auszuführen.

Von dieser Möglichkeit soll in der vorliegenden Arbeit jedoch kein Gebrauch gemacht werden, vielmehr werden hier Referenznetze ausschließlich für die Visualisierung von

Konzepten verwendet. Dieser Abschnitt beschreibt, wie die gezeigten Modelle zu lesen sind, wobei eine objektorientierte Sichtweise eingenommen wird.

Stellen beinhalten die Daten, mit denen das betreffende Netz arbeitet. Grundsätzlich können Objekte eines beliebigen Typs auf einer Stelle liegen, etwa Javaobjekte, primitive Typen oder andere Netze. Die Modelle in dieser Arbeit verwenden, wenn nicht anders angegeben, die letzte Variante. Vereinfacht können Stellen als Variablen betrachtet werden.

Transitionen, Uplinks und Downlinks Transitionen sind Elemente, die eine Aktion des Netzes repräsentieren. Uplinks eignen sich dazu, das Schalten der dazu gehörigen Transition auszulösen. Daher können solche Transitionen im Sinne der Objektorientierung als Methode des Netzes gesehen werden. Ein Downlink wiederum ist ein Methodenaufruf einer Netzinstanz, die sich auf einer Stelle befindet.

2.3 Zusammenfassung

Dieses Kapitel stellt die beiden in der vorliegenden Arbeit verwendeten Modellierungstechniken vor, die *Unified Modelling Language* UML einerseits, *Referenznetze* andererseits.

Bei der UML handelt es sich um einen von der *Object Management Group* OMG [Obj03a] gepflegten Standard. Sie hat sich für die Modellierung objektorientierter Software etabliert und findet breite Anwendung. Sie enthält verschiedene Diagrammtypen, mit denen jeweils statische oder dynamische Aspekte eines Systems dargestellt werden können.

Referenznetze [Kum02] eignen sich ebenfalls für die Modellierung von Softwaresystemen. Sie stellen eine Erweiterung der Petrinetze dar, in dem sie objektorientierte Konzepte, wie Exemplare von Netzen, einführen.

Die Vorteile der UML sind ihre Übersichtlichkeit und ihre weite Verbreitung. Dadurch erleichtert ihre Verwendung die Kommunikation zwischen Projektmitarbeitern.

In Referenznetzen erstellte Modelle hingegen können simuliert werden und bieten daher dem Entwickler schon zur Entwurfszeit die Möglichkeit, Fehler festzustellen. Durch ihre Fundierung auf Petrinetzen sind sie besonders gut geeignet, nebenläufige Systeme darzustellen.

Alles in allem ist es von dem zu entwerfenden System abhängig, welche der beiden Modellierungstechniken gewählt werden sollte. Auch in dieser Arbeit werden – je nach Bedarf – beide verwendet.

Kapitel 3

Komponenten

Dieses Kapitel führt das Konzept der Softwarekomponenten ein, die eine Möglichkeit darstellen, Software übersichtlicher und erweiterbarer zu gestalten. Dazu wird zunächst ein Überblick über das Konzept der komponentenorientierten Softwareentwicklung gegeben.

Da sich in der Literatur verschiedene Definitionen des Begriffs *Komponente* finden, soll zuerst geklärt werden, wie er in der vorliegenden Arbeit verwendet wird. Dazu werden im ersten Abschnitt dieses Kapitels die grundlegenden Begriffe definiert und die Vor- und Nachteile der komponentenorientierten Entwicklung diskutiert.

Im weiteren Verlauf werden die in *Renew* fest gestellten Anforderungen beschrieben und die Frage erörtert, ob und wie diese durch die Einführung von Komponenten gelöst werden können.

Anschliessend folgen zwei Beispiele für existierende Komponentensysteme; diese dienen dazu, die vorher bestimmten Begriffe und Konzepte zu erläutern. Die Merkmale, die bei dieser Analyse identifiziert werden, fließen später in das Komponentenmodell, das *Renew* zu Grunde liegen soll, ein. Darauf aufbauend wird im nächsten Kapitel eine spezielle Art der Komponente eingeführt, das Plugin.

3.1 Begriffe

Bevor die Diskussion über Komponenten beginnen kann, ist es unerlässlich, die zu Grunde liegenden Begriffe zu klären. Dies ist keine leichte Aufgabe, da sich in der Literatur sehr unterschiedliche Definitionen dafür finden, was eine Komponente ausmacht.

Dabei werden die einzelnen Aspekten der vorgefundenen Definitionen zu der Bestimmung des Begriffs *Komponente* zusammengesetzt, wie er für die vorliegende Arbeit sinnvoll ist.

3.1.1 Komponenten

Sametinger führt Komponenten als Einheiten der Wiederverwendung von Software ein [Sam96]. In seiner Definition ist dieser Aspekt weniger ausgeprägt:

> A component is a self-contained, clearly identifiable piece that describes and/or performs specific functions and has clear interfaces, appropriate documentation and a defined reuse status.

Der *reuse status* bezeichnet hier Informationen darüber, wer die Komponente entwickelt hat, für die Wartung verantwortlich ist etc. Die Formulierung *describes and/or performs* deutet darauf hin, dass es sich bei Komponenten auch um reine Dokumentationsartefakte handeln kann. Diese Sichtweise gilt für die vorliegende Arbeit nicht: Komponenten sollen immer ausführbaren Programmcode zusammen mit der darauf bezogenen Dokumentation enthalten.

Außerdem schränkt der Autor selbst die Anforderung ein, eine Komponente müsse *self-contained* sein, also für sich selbst stehen können: so könne Funktionalität auch ausgelagert werden, dabei sei es jedoch notwendig, die dabei entstehenden Abhängigkeiten zu dokumentieren.

Diese Aussage ist wichtig, da sie eine wichtige Ergänzung der Definition darstellt. Tatsächlich wird in Teil II, in dem die Integration dynamischer Komponenten in *Renew* beschrieben wird, ersichtlich, dass die Abhängigkeitsverwaltung einen nicht zu vernachlässigenden Entwurfsaufwand beansprucht.

Griffel [Gri98], [Gri01] betrachtet eine Komponente sinngemäß als eine Aggregation von Objekten, durch die eine semantisch bestimmte Fachdomäne abgedeckt wird. Die Komponente kann dabei mit anderen kommunizieren, um ihre Aufgabe zu lösen.

Hier sind zwei Aspekte interessant: zunächst spielt die Betrachtung, dass Komponenten aus mehreren Klassen zusammengesetzt werden, eine Rolle. Wie später zu sehen sein wird, werden Komponenten bei *Renew* sogar die in Java zur Bündelung von Klassen verwendeten *package*s überspannen. Zum anderen ist die Betonung der Fachlichkeit von Komponenten wichtig. Dies ist ein wichtiges Kriterium bei der späteren Komponentenbildung für *Renew* (siehe Kapitel 8.3).

Als letzte Definition von *Komponente* sei die von Eichler [Eic02] erwähnt, der mit Einschränkungen die Auffassung von Szyperski [Szy02, S. 41] verwendet: für ihn ist eine Komponente eine

> Einheit der Verteilung mit klar definierten Schnittstellen und deutlich gekennzeichneten [unidirektionalen] Abhängigkeiten.

Tatsächlich fasst diese Begriffsbestimmung viele Aspekte zusammen, die auch in der vorliegenden Arbeit übernommen werden können. Zu den bereits besprochenen Punkten kommt die Forderung hinzu, dass die Abhängigkeiten unidirektional sein müssen, zwei Komponenten also nicht gegenseitig voneinander abhängig sein dürfen.

Unter Berücksichtigung aller bisherigen Alternativen kann also nun eine Definition des Komponentenbegriffs entstehen, die die für diese Arbeit maßgeblichen Aspekte abdeckt.

Komponente Eine Komponente ist eine Einheit der Verteilung, die aus ausführbarem Code und dazu gehöriger Dokumentation besteht und domänenspezifische Dienste zur Verfügung stellt.

Verschiedene Komponentensysteme unterscheiden sich darin, was eine Komponente anbieten muss, um in dem jeweiligen System gültig zu sein; etwa, wie die Dienste angeboten werden und wie die Schnittstellen auszusehen haben. Daher ist eine Komponente auf ein Komponentensystem festgelegt.

3.1.2 Dienste und Schnittstellen

Die Begriffe *Dienst* und *Schnittstelle* in der oben dargelegten Definition verdienen eine genauere Betrachtung. Szyperski nennt Schnittstellen "*means, by which components connect*" [Szy02, S. 50]; Griffel sagt, eine Schnittstelle sei die "Grenze einer Softwarekomponente" [Gri98, S. 53]. Diese Begriffsbestimmungen sehen Schnittstellen als technische Konstrukte, die den Zugriff auf die Funktionalität einer Komponente ermöglichen.

Griffel jedoch erweitert dieses Verständnis, indem er fest stellt:

> Die Schnittstelle stellt aber gewissermaßen den "Vertrag" bereit, den ein [...] Benutzer [...] zur Nutzung [...] einer Komponente eingehen kann [Gri98, S.54].

Laut Szyperski ist ein solcher Vertrag hingegen nicht *Bestandteil* der Schnittstelle, sondern lediglich damit *verbunden* (*attached to* [Szy02, Seite 500]) und könne syntaktische, auf die Dienstgüte (*quality-of-service*) bezogene sowie semantische Aspekte beinhalten.

Für das Komponentenverständnis der vorliegenden Arbeit ist dabei besonders der Aspekt der Semantik von Bedeutung. Dieser ergibt sich aus der fachlichen Domäne und Funktion der zugehörigen Komponente.

Um die Trennung zwischen der Schnittstelle einer Komponente, die den *Zugriff* auf Funktionalität erlaubt, und ihrer fachlichen Bedeutung auszudrücken, verwendet diese Arbeit die Begriffe *Dienst* und *Schnittstelle*.

Dienst Ein Dienst ist eine fachliche Funktionalität, die von einer Komponente zur Verfügung gestellt wird. Die Spezifikation eines Dienstes beschreibt auf semantischer Ebene das Verhalten der Komponente.

Schnittstelle Eine Schnittstelle ist ein Konstrukt, mit dem eine Komponente einen Dienst bereit stellt, so dass andere Komponenten darauf zugreifen können.

Die Komponenten-Verwaltung verwaltet die Dienste, die durch vorhandene Komponenten bereitgestellt werden. Wenn eine Komponente einen Dienst benötigt, so liefert die Verwaltung ihr die entsprechende Schnittstelle einer Komponente, die diesen Dienst anbietet. Die Verwendung einer Schnittstelle durch eine andere Komponente heisst *Kommunikation*:

Kommunikation Kommunikation ist der Vorgang zwischen zwei Komponenten, bei denen die eine Dienste der anderen nutzt.

Bei einer Kommunikation kann ein Datenaustausch zwischen den beiden Komponenten stattfinden, es kann sich aber auch um einen reinen Benachrichtigungsmechanismus handeln.

3.1.3 Abhängigkeit

Oft enthält eine Komponente den gesamten Programmcode, der zu ihrer Ausführung nötig ist. Das ist jedoch nicht immer der Fall, da es dazu führen würde, dass häufig verwendete Funktionen in mehreren Komponenten vorhanden sein müssten. Dies widerspricht dem Wiederverwendungsgedanken von Komponenten (vgl. Abschnitt 3.2.1) und kann dadurch vermieden werden, dass der entsprechende Code in eine eigene Komponente ausgelagert wird [Sam96]. Dadurch sind die ursprünglichen Komponenten abhängig von der neuen Komponente.

Abhängigkeit Eine Komponente **K** ist von einer Komponente **L** abhängig, wenn **K** zu ihrer Ausführung Dienste von **L** benötigt. Abhängigkeit ist unidirektional, d.h. wenn **K** abhängig von **L**, darf **L** nicht abhängig von **K** sein.

Abhängigkeiten müssen in der Dokumentation von Komponenten angegeben werden.

3.1.4 Dynamik

Ein weiterer Aspekt von Komponenten ist der der Dynamik [Eic02]. Statische Komponenten werden zur Entwicklungszeit verwendet und zu einem Softwaresystem zusammengefügt. Als dynamische Komponenten werden diejenigen bezeichnet, die zur Laufzeit in das System eingefügt werden und ihre Funktionalität zur Verfügung stellen können, wodurch sie wesentlich flexibler einsetzbar sind.

Besonders bei der Verwendung von dynamischen Komponenten ist es wichtig, auf ihre Abhängigkeiten zu achten. Das System, das die Komponenten lädt, muss dafür Sorge tragen, dass nur Komponenten zum Einsatz kommen, deren Abhängigkeiten erfüllt sind. Dies spielt im Falle dynamischer Komponenten nicht nur beim Start des Systems eine Rolle, sondern auch zur Laufzeit.

3.1.5 Abgrenzung zwischen Objekten, Komponenten und Agenten

Mit der Definition des Komponentenbegriffes stellt sich die Frage, wie sich die Komponente von Konzepten wie dem des *Objekts* aus der objektorientierten Programmierung oder dem des *Agenten* [Röl99], [Jen00] unterscheidet.

Objekte

Auf den ersten Blick erfüllt auch das Objekt die gegebene Definition: Objekte sind Einheiten der Verteilung, die ausführbaren Code enthalten und Dienste anbieten. Auch Dokumentation kann von Objekten zur Verfügung gestellt werden. Dennoch gibt es Unterschiede zwischen objekt- und komponentenorientierten Systemen.

Zum einen handelt es sich dabei um einen strukturellen Unterschied. In [Eic02, Seite 27f] findet sich der Hinweis, Komponenten seien eine gröbere Einteilungseinheit als Objekte; wenn Komponenten mit einer objektorientierten Sprache implementiert sind, können sie aus mehreren Objekten zusammengefasst sein. Dies ist jedoch kein konzeptioneller Unterschied, und in der Tat ist es möglich, dass ein einzelnes Objekt die Dienste im Sinne einer Komponente anbietet.

Zudem lassen sich von Objekten beliebig viele Exemplare erzeugen. Dies ist für Komponenten im hier verwendeten Sinne nicht möglich. Dies würde lediglich bereits vorhandene Funktionalität duplizieren. Andererseits gibt es aber auch Klassen, von denen lediglich

ein Exemplar erzeugt werden kann: dies ist etwa bei dem Entwurfsmuster *Singleton* der Fall [GHJV95]. In der Tat wird dieses Entwurfsmuster bei dem späteren Komponentensystem für *Renew* die Rolle spielen, dass jeweils ein Singleton-Objekt eine Komponente repräsentiert (siehe Abschnitt 7.2.1).

Unterschiedlich ist jedoch die Art und Weise, wie das Auffinden von Diensten stattfindet. In objektorientierten Systemen kennt ein Objekt diejenigen Partnerobjekte, die es zur Erfüllung seiner Aufgaben benötigt.

In Komponentensystemen hingegen gibt es einen Vermittler, der die Dienste und Komponenten, die diese anbieten, verwaltet. Benötigt eine Komponente einen Dienst, fragt sie diese Verwalterinstanz nach Komponenten, die den gewünschten Dienst zur Verfügung stellen. Zu diesem Zweck gibt es eine Schnittstelle, die jede Komponente in einem gegebenen System erfüllen muss und über die die zur Verfügung gestellten Dienste abgefragt werden können.

Was ist nun, wenn es ein Objekt gibt, das diese Schnittstelle erfüllt und auch die anderen oben genannten Vorraussetzungen erfüllt? Die Antwort darauf lautet, dass es sich dann bei diesem Objekt auch um eine Komponente im Sinne des entsprechenden Komponentensystems handelt. Es ist dann einerseits ein Objekt, weil es in einer objektorienten Sprache entwickelt wurde, andererseits eine Komponente, weil es Dienste gemäß dem Komponentensystem anbietet.

Agenten

Das Konzept eines Dienstevermittlers gibt es auch in der agentenorientierten Softwareentwicklung. Die von der FIPA [Fou03] festgelegte Multi-Agenten-Architektur sieht einen so genannten *Directory Facilitator* (DF) vor, der eben diese Funktion erfüllt.

Was unterscheidet also einen Agenten von einer Komponente? Um diese Frage zu beantworten, ist es sinnvoll, eine Liste der Merkmale von Agenten zu betrachten.

Die Definition in [Jen00] besagt, ein Agent sei

> ein gekapseltes Computersystem das in einer Umwelt flexibel und eigenständig Aktionen ausführen kann, um seinen Entwurfszweck zu erfüllen.

Dabei bedeutet *gekapselt*, dass klar definierte Grenzen und Schnittstellen vorhanden sind, was auch bei Komponenten der Fall ist.

Die *Umwelt* kann von Agenten durch Sensoren wahrgenommen und durch Effektoren beeinflusst werden. Im weiteren Sinne trifft auch dies auf Komponenten zu, da sie etwa Dienste anbieten und nach angebotenen Diensten fragen können.

Die von Agenten geforderte *Flexibilität* bedeutet, dass Agenten nicht an ein bestimmtes, vorgegebenes Verhalten gebunden sind, sondern je nach ihrer Umgebung unterschiedlich reagieren können. Dieser Punkt wird von Komponenten nicht erfüllt, da sie nicht wie Agenten auf Veränderungen der Umwelt reagieren können. Es wäre sicher denkbar, intelligente Komponenten zu entwickeln, dies ist jedoch nicht das Hauptziel von komponentenorientierten Systemen.

Eigenständigkeit ist bei Komponenten ebenfalls nicht gegeben. Auch hierfür ist eine Form von Intelligenz nötig, die Komponenten fehlt. Eigenständigkeit ist im Gegenteil sogar in Komponenten nicht wünschenswert, da sich autonome Komponenten dafür entscheiden könnten, einen bereit gestellten Dienst nicht auszuführen. Dies würde jedoch bedeuten, dass der zum Dienst gehörige Vertrag gebrochen würde. Eine solche Entwicklung ist nicht im Sinne der Komponentenorientierung, die die Zuverlässigkeit von Komponenten als eines der Entwicklungsziele betrachtet.

Es kann also gesagt werden, dass eine Komponente kein Agent ist. Wie sieht es mit der Gegenrichtung aus? Ist ein Agent eine Komponente? Agenten erfüllen in der Tat alle Vorraussetzungen, die eine Komponente ausmachen. Insbesondere das Konzept der Dienste, die über eine Verwaltungsinstanz bekannt gemacht werden, sollte es Agenten ermöglichen, sich wie Komponenten zu verhalten. Dazu muss ein Agent lediglich die oben geforderte Schnittstelle erfüllen. Wie für das Objekt gilt auch, dass ein Agent, der seine Dienste so anbietet, wie es das Komponentensystem spezifiziert, als Komponente zu verstehen ist.

Agenten werden *mobil* genannt, wenn sie eine Agentenplattform betreten und wieder verlassen können. Solche Agenten als Komponenten einzusetzten ist nur bei Verwendung eines dynamischen Komponentensystems sinnvoll, damit es möglich ist, ihn zur Laufzeit einzubinden und wieder zu entfernen.

3.2 Verwendung von Komponenten

Nachdem die grundlegenden Begriffe geklärt sind, soll nun diskutiert werden, warum und unter welchen Umständen die Softwareentwicklung mit Komponenten überhaupt sinnvoll ist.

Als Nachteil der Verwendung von Komponenten ist der Overhead zu erwähnen, der bei der Entwicklung entsteht. Schon bei der objektorientierten Programmierung wird erheblicher zeitlicher Aufwand in Programmdokumentation und Schnittstellenentwurf gesteckt; dies auf einer weiteren Ebene noch einmal zu tun, erscheint zunächst wenig sinnvoll oder sogar kontraproduktiv.

Dies hängt jedoch von der Größe des zu entwickelten Systems ab. Des weiteren spielt eine Rolle, inwiefern das fachliche Gebiet, für das die Anwengung entwickelt werden soll, eine Aufteilung in Domänen zulässt.

Auf der anderen Seite bietet die Verwendung von Komponenten ab einer gewissen Projektgröße verschiedene Vorteile. Viele davon resultieren daraus, dass Komponenten die Wiederverwendung von Code unterstützen. Eine ausführliche Erörterung unter verschiedenen Blickwinkeln nennt Sametinger [Sam96]; hier sollen nur die wichtigsten Aspekte angesprochen werden.

3.2.1 Wiederverwendung

Als erstes sollen die Vorteile der Wiederverwendung von Komponenten genannt werden. Diese werden in [Sam96] in die drei groben Kategorien *Qualitätsverbesserung*, *Aufwandsreduzierung* und *andere Vorteile* eingeteilt.

- Unter *Qualitätsverbesserung* fallen demnach folgende Punkte:

  Qualität der wiederverwendeten Komponenten Dies resultiert daraus, dass Fehler in einer häufig verwendeten Komponente eher gefunden werden und somit schneller verbessert werden.

  Produktivität in der Entwicklung Dadurch, dass schon vorhandene Funktionalität nicht mehrfach entwickelt werden muss, können die Entwickler sich mit anderen Aufgaben beschäftigen.

  Performanz des erstellten Systems Dies wird einerseits dadurch erreicht, dass der Aufwand, eine mehrfach verwendete Komponente zu optimieren, sich eher lohnt und mehr Vorteile bringt. Andererseits schont die Tatsache, dass Code nicht mehrfach geladen wird, Ressourcen.

  Interoperabilität zwischen verschiedenen Systemen Die Verwendung gleicher Komponenten in verschiedenen Programmen vereinfacht den Datenaustausch, da dies die Gefahr minimiert, dass die beiden Kommunikationspartner inkompatible Protokolle verwenden.

- Die *Aufwandsreduzierung* beinhaltet:

  Vermindern redundanter Arbeit Dadurch, dass viel Programmierarbeit nicht mehrfach erledigt werden muss, wird Zeit gespart, die in andere Aufgaben investiert werden kann. Die Verwendung von Komponenten unterstützt dies, da durch sie schon vorhandener Code wiederverwendet wird.

  Beschleunigung der Entwicklung Da bei der Entwicklung Zeit gespart wird, können Projekte schneller beendet werden.

Verminderung des Dokumentationsaufwandes Da sich die Dokumentation eines aus Komponenten erstellten Systems auf die neu entwickelten Teile sowie die Kommunikation zwischen den verwendeten Komponenten beschränkt, ist weniger neue Dokumentation zu erstellen.

Verminderung der Wartung Es ist leichter, Systeme, die aus Komponenten zusammengesetzt werden, zu warten. Dies liegt daran, dass die Qualität der Komponenten bereits hoch ist und sie getrennt vom daraus konstruierten System gewartet werden. Somit wirkt sie sich auf verschiedene Systeme aus, in denen die entsprechenden Komponenten eingesetzt werden.

Niedrigere Einarbeitungszeit Entwickler, die neu in ein Projekt kommen, müssen sich lediglich mit den Schnittstellen der Komponenten beschäftigen, die sie auch verwenden. Dadurch können sie schneller produktiv an dem System mitarbeiten.

- Unter *andere Vorteile* versteht SAMETINGER:

  Schnelle Prototypentwicklung Da viele grundsätzliche Bestandteile eines Systems als Komponenten vorliegen, ist die Produktion einer vorläufigen, aber bereits lauffähigen Vorabversion schneller zu bewerkstelligen.

  Weitergabe von Designerfahrung Neue Entwickler, die an dem Produkt arbeiten, profitieren von dem Einsatz von Komponenten, da sie einen Einblick in die zu Grunde liegenden Entwurfsentscheidungen erhalten.

All diese Aspekte beziehen sich auf die Verwendung von statischen Komponenten, die bei Entwicklung zu einem System zusammengesetzt werden. Bei dynamischen Komponenten kommt noch der Vorteil der Flexibilität hinzu. Als Konsequenz wird *Qualitätsverbesserung* um diesen Punkt ergänzt:

Austausch fehlerhafter Programmteile macht es möglich, Komponenten, bei denen ein Fehler festgestellt wurde, durch korrigierte zu ersetzen, ohne das Programm beenden zu müssen.

Zusätzlich zum Ersetzen bestehender Teile können auch noch nicht vorhandene Teile ergänzt werden:

Erweiterbarkeit des laufenden Systems. Funktionalität, die zunächst nicht vorgesehen war, kann durch das Einbringen neuer Komponenten hinzugefügt werden.

Insgesamt gibt es also gute Argumente für die Wiederverwendung von Programmcode. Diese wird durch den Einsatz von Komponenten unterstützt.

3.2.2 Konfigurationsmanagement

Der Begriff *Konfigurationsmanagement* kommt aus dem Projektmanagement. Konfigurationsmanagement beschäftigt sich mit der Verwaltung aller Artefakte im Entwicklungsprozess.

Dart [Dar00] beschreibt Konfigurationsmanagement als "Disziplin der Steuerung der Evolution eines Softwaresystems"; dazu gehören laut [Ber91]:

> Identifikation, Kontrolle, Zustandsdokumentation und Prüfung eines Softwareprodukts, während es sich von der Konzeptionsphase über Auslieferung und Wartung weiterentwickelt.

Der IEEE-Standard 729-1983 definiert:

> Konfigurationsmanagement ist der Prozess, alle Komponenenten eines (Software-) Systems zu identifizieren und zu definieren, die Freigaben und Änderungen zu dokumentieren, und die Vollständigkeit und Korrektheit der Komponenten zu verifizieren.

In dieser Definition wird der Begriff *Komponente* verwendet. Es ist jedoch wichtig zu bemerken, dass er hier in einem anderen Zusammenhang und mit einer anderen Bedeutung als im letzten Abschnitt definiert ist. Dennoch lassen sich die dort beschriebenen Softwarekomponenten als die Verwaltungseinheiten im Sinne des Konfigurationsmanagements verwenden.

Dazu findet sich in [CM 03] die Definition von *Komponentenmodell*:

> Versionskontrollmodell, bei dem als wichtigstes Strukturierungsmittel eine Systembeschreibung die Grundlage für Versionskontrolle, insbesondere die Erzeugung von Konfigurationen, und Produktionsmanagement bildet.

Eine Komponente bildet hier also ausschließlich eine Strukturierungseinheit für die Verwaltung eines Softwareproduktes und beschäftigt sich nicht mit dem in dieser Arbeit hauptsächlich thematisierten Verhalten zur Laufzeit.

Diese beiden Konzepte stehen aber keineswegs im Wiederspruch zueinander, im Gegenteil: Die fachliche Aufteilung, die durch die Verwendung von Komponenten in das System eingebracht wird, unterstützt das Konfigurationsmanagement, da keine künstliche Partitionierung eines monolithischen Systems erfolgen muss.

Als Ziele des Konfigurationsmanagements werden angesehen [CM 03]:

- Qualitätssteigerung durch standardisierte Abläufe in der Entwicklung und Automatisierung von Routineaufgaben.
- Transparenz der Entwicklung, also eine Übersicht, wer was geändert hat; dies wird durch die Verwendung von Versionsverwaltungssystemen erreicht.
- Kostenreduktion und Zeitersparnis durch Automatisierung von Standardprozessen.

Dies sind grundsätzlich für jedes Softwareprodukt wünschenswerte Eigenschaften. Dabei haben sich im Konfigurationsmanagement für die Durchsetzung dieser Anforderungen folgende Untergebiete etabliert:

Änderungsmanagement Dieser Bereich beschäftigt sich damit, jede an der Software vorgenommene Änderung wie Fehlerkorrekturen oder Weiterentwicklungen zu verwalten. Darunter fällt auch die Festlegung der erwünschten Weiterentwicklung des Produktes. In streng organisierten Projekten dürfen lediglich vorher angeforderte Änderungen unternommen werden.

Problemmanagement Unter diesem Punkt werden alle Aktivitäten gefasst, die das Auffinden und Beheben von Fehlern im zu entwickelnden System betreffen.

Produktionsmanagement Dieses beschäftigt sich mit der Zusammenstellung aller Verwaltungseinheiten, die im Konfigurationsmanagement festgelegt sind, zum Endprodukt.

Versionsmanagement Im Versionsmanagement wird die Planung neuer Produktversionen sowie die Verwaltung der vorhandenen Versionen durchgeführt.

Zur Unterstützung des Konfigurationsmanagements gibt es für jeden der oben genannten Punkte Softwarewerkzeuge, die bei den jeweiligen Aufgaben behilflich sind.

3.2.3 Nachteile von Komponenten

Bisher sind lediglich Argumente *für* den Einsatz von Komponenten genannt worden. Diese Vorteile haben jedoch ihren Preis, daher sind Komponenten nicht für jedes Projekt geeignet. Die Entscheidung, ob Komponenten verwendet werden sollen oder nicht, ist daher von Fall zu Fall zu treffen. Dieser Abschnitt beschreibt die Nachteile, mit denen zu rechnen ist.

Gefahr der Inkompatibilität

Wie Griffel erläutert [Gri01, S. 124f], können beim Einsatz von Komponenten Inkompatibilitäten zwischen den einzelnen Systemteilen entstehen. Dabei unterscheidet er

- syntaktische Inkompatibilitäten, die entstehen, wenn inkompatible Nachrichten ausgetauscht werden, etwa solche mit unterschiedlichen Parametertypen, und
- semantische Inkompatibilitäten, die dadurch zu Stande kommen, dass eine versendete Nachricht falsch interpretiert wird.

Die Gefahr der Inkompatibilität ist dann besonders groß, wenn Komponenten verschiedener Hersteller verwendet werden.

Erhöhter Entwicklungsaufwand

Wie bereits eingangs erwähnt, ist der Einsatz von Komponenten zunächst mit einem Entwicklungsoverhead verbunden. Auch hier muss von Projekt zu Projekt unterschieden werden, ob dieser Aufwand gerechtfertigt ist.

So steigt mit jeder Wiederverwendung einer Komponente der Nutzen, sie erstellt zu haben. Werden Komponenten primär zum Konfigurationsmanagement eingesetzt, ist es schwieriger, den dadurch erzielten Gewinn zu bestimmen.

Festgelegte Schnittstellen

Dieser Punkt trifft zu, wenn Dritthersteller Komponenten verwenden. In diesem Fall verlassen sich diese darauf, dass die von ihnen eingesetzten Komponenten über eine bestimmte Schnittstelle verfügen. Dadurch wird es problematisch, die Schnittstellen in der späteren Entwicklung der Komponenten zu verändern.

Im Idealfall sind Schnittstellen stabil, und nur die Implementation, mit der sie umgesetzt werden, kann sich ändern. In der Realität ist es jedoch oft so, dass, besonders bei großen Änderungen im System, auch die Schnittstellen angepasst werden müssen.

Dies kann bei einem monolitischen System, bei dem sich die Objektschnittstellen ändern, zwar aufwändig umzusetzen sein; verändern sich jedoch Komponentenschnittstellen, ist es schwierig nachzuvollziehen, welche externen Benutzer diese Schnittstellen verwenden. Daher sollten die Schnittstellen von Komponenten mit großer Sorgfalt entworfen werden.

3.3 Eignung für *Renew*

Es soll nun geprüft werden, ob die Verwendung der Komponentenorierung für eine Neuausrichtung bei der Architektur von *Renew* sinnvoll ist. Dabei werden die in der Einleitung aufgeführten Anforderungen der Verwaltbarkeit, Konfigurierbarkeit und Erweiterbarkeit durchgegangen und geprüft, inwiefern eine Umstellung von *Renew* auf Komponenten den jeweiligen Punkt erfüllt.

3.3.1 Verwaltbarkeit der Software und des Sourcecodes

Um zu sehen, ob Komponenten helfen, *Renew* übersichtlicher und verwaltbarer zu machen, hilft ein Blick auf den bisherigen Zustand von *Renew* unter diesem Gesichtspunkt. Da es sich bei *Renew* um eine objektorientiert in der Sprache Java entworfene Software handelt, stehen zwei einfache Metriken zur Verfügung.

Als erstes wäre die Anzahl der Klassen zu nennen. Zum Zeitpunkt des Beginns der vorliegenden Arbeit bestand *Renew* aus etwa 1100 Klassen; in dieser Angabe sind nur Top-Level-Klassen enthalten, also keine inneren oder anonymen. Ein Projekt dieser Größenordnung lässt sich unmöglich verwalten, ohne eine weitere Hierarchisierung vorzunehmen.

Zweitens bietet sich die Anzahl der *package*s als Maßstab an. Java bietet auf Ebene der Programmiersprache die Möglichkeit, Klassen in so genannten *package*s zu bündeln. Hierbei werden mehrere zusammengehörige Klassen zusammen gefasst; dadurch wird, neben der Bildung von Namensräumen auf der technischen Seite, auch mehr Übersichtlichkeit eingeführt. *Renew* enthielt zu Beginn dieser Arbeit 16 *package*s, die das Kernsystem sowie einige später hinzugekommene Erweiterungen beinhalten. Hinzu kommen sieben aus dem für die GUI verwendeten Grafikframework *JHotDraw* sowie das *collections-package* von Doug Lea, insgesamt also 24.

Dies ist schon überschaubarer. Die Verwendung von *package*s allein hat jedoch einige Nachteile. Zum einen ist die Aufteilung von Klassen sehr unregelmäßig: die vorhandenen *package*s enthalten von unter 10 bis zu über 90 Klassen. Dies wird sich zwar unter Umständen auch bei einer komponentenbasierten Aufspaltung nicht vermeiden lassen; hierbei werden jedoch insbesondere mit Hilfe von Dokumentation zusätzliche Maßnahmen möglich, dieses Problem anzugehen.

Ein weiteres Problem liegt in der Komplexität der Software begründet. Für Entwickler, die sich mit dem System noch nicht auskennen, ist es schwierig zu entscheiden, an welcher Stelle sie eine geplante Verbesserung anbringen sollen. Dies ist besonders bei *Renew* häufig der Fall, da an der Universität Hamburg regelmäßig Veranstaltungen angeboten werden, in denen Studenten Erweiterungen für *Renew* programmieren.

Der Grund für den Mißstand liegt darin, dass es in *Renew* keine Dokumentation für die einzelnen *package*s gibt. Hier würde auch ein einheitliches Vorgehen zur Dokumentation der *package*s, ihrer allgemeinen Funktion sowie der wichtigsten Klassen Abhilfe schaffen. Sinnvoll wäre etwa die Verwendung des javadoc-Systems von Java: die Pflege einer javadoc-Datei für jedes *package* würde sich anbieten. Bei der Entwicklung von *Renew* wird javadoc zwar verwendet, jedoch nur auf Klassenebene.

Der dritte Nachteil von *package*s bei der Codeverwaltung ist, dass Abhängigkeiten nicht expliziert werden: es ist nicht ersichtlich, auf welche *package*s sich die Änderungen an bestimmten Klassen auswirken. Daher müssen im Zweifelsfall alle Klassen neu kompiliert werden, um sicher zu stellen, dass keine Inkonsistenzen im System auftreten.

Dadurch, dass jede Komponente bereits die Information enthält, von welchen anderen sie abhängig ist, wird diesem Problem vorgebeugt: so müssen bei Änderungen einer

Komponente nur diejenigen neu übersetzt werden, die darauf basieren.

Als Ergebnis lässt sich sagen, dass die Komponentisierung von *Renew* ein Erfolg versprechender Weg ist, die zunehmende Zahl von Klassen in Hinsicht auf Quellcodeverwaltung in den Griff zu bekommen.

3.3.2 Konfigurierbarkeit

Während unter *Konfigurationsmanagement* der Schritt vom Quellcode in das laufende System zu verstehen ist, bezeichnet der Begriff *Konfigurierbarkeit* die Möglichkeit für den Benutzer, das bereits ausgelieferte System seinen Bedürfnissen anzupassen.

Als wichtigste Konfigurationsmethode ermöglicht es *Renew*, durch Setzen der Umgebungsvariable *de.renew.mode* zu steuern, welchen Compiler das System zur Übersetzung der vom Benutzer erstellten Netze verwenden soll. Dabei ist der *Mode* noch für einige andere Dinge zuständig; er kann der graphischen Oberfläche zusätzliche Menüeinträge und neue Zeichenelemente hinzufügen sowie die Anzeige der Inhalte von Stellen während der Simulation steuern. An dieser Stelle ist strukturell eine unschöne Verschmelzung von fachlichen Konzepten, eben der Wahl des Compilers, mit der Unterstützung der graphischen Oberfläche zu bemängeln. Zudem ist es zur Laufzeit nicht möglich, den gewählten *Mode* zu ändern.

Besonders in dieser Hinsicht verspricht die Verwendung von dynamischen Komponenten Besserung. Damit ist es möglich, die *Mode*s als Komponenten zu modellieren und bei Benutzung des Systems auszuwählen.

3.3.3 Erweiterbarkeit

Der letzte Punkt der in der Einleitung beschriebenen Anforderungen ist der der Erweiterbarkeit. Ein System ist dann als erweiterbar zu bezeichnen, wenn es möglich ist, zusätzliche Funktionalität hinzuzufügen. An dieser Stelle kann unterschieden werden, ob dies zur Kompilier- oder zur Laufzeit möglich ist.

Da der Quellcode von *Renew* zur Verfügung steht, ist es zur Kompilierzeit durchaus erweiterbar: Veränderungen können am Code vorgenommen und einkompiliert werden. Hieran ist zu bemängeln, dass dadurch genau die Probleme geschaffen werden, die durch das in Abschnitt 3.2.2 beschriebene Konfigurationsmanagement verhindert werden sollen: inkompatible Programmversionen, unbemerkt hinzukommende Fehler, nicht von allen Benutzern verwendete Funktionen, die sowohl überflüssige Ressourcen verwenden als auch die Benutzbarkeit verringern.

Dem ist mit der Komponentisierung von *Renew* gut beizukommen. Dabei ist die Idee, dass sich ein Benutzer, der eine neue Funktionalität benötigt, diese in einer Komponente bereitstellt. Da dadurch keine bereits bestehenden Komponenten betroffen sind, wird der Lauffähigkeit des Systems kein Abbruch getan.

Andererseits ermöglicht es das Komponentenkonzept nicht, durch neue Komponenten das Verhalten bereits vorhandener Komponenten zu verändern: da die Abhängigkeit zwischen zwei Komponenten immer unidirektional ist, kann zwar die erweiternde Komponente von der zu erweiternden abhängen, also ihre Schnittstellen kennen und dadurch ihre Dienste nutzen; in der anderen Richtung ist dies jedoch nicht möglich.

Die Lösung dieses Problems besteht darin, die Schnittstellen der zu erweiternden Komponente so zu entwerfen, dass sie auf die Funktionalität der hinzuzufügenden Komponente zugreifen kann, ohne deren Struktur genau zu kennen. Dies ist für die vorliegende Arbeit das kennzeichnende Merkmal von Plugin-Fähigkeit. Mit dem Konzept der Plugins beschäftigt sich Kapitel 4.

3.4 Existierende Komponentenarchitekturen

Die Vorteile der Softwareentwicklung mit Hilfe von Komponenten sind zum einen Teil technischer, zum anderen aber auch wirtschaftlicher Natur: Zeitersparnis etwa bedeutet für ein Unternehmen einerseits, die Entwicklungskosten gering zu halten, andererseits wird eine kürzere *time-to-market* erreicht. Dies bedeutet, dass das Produkt zu einem früheren Zeitpunkt verkauft und somit mehr Marktanteile gesichert werden können.

Aus diesem Grund hat sich der Komponentenansatz auch in der Wirtschaft durchgesetzt, und viele Unternehmen bieten Lösungen für die komponentenorientierte Entwicklung an. Insbesondere seit dem großen Erfolg des Internets hat sich dabei der Fokus dahingehend verschoben, dass Komponenten nicht nur als Softwarebausteine gesehen werden, die sich zu einer Applikation zusammensetzen lassen, sondern dass sie als verteilte Dienstanbieter von vielen Rechnern im Netzwerk genutzt werden können.

In dieser Kategorie von Komponentensystemen ist die *Common Object Request Broker Architecture* CORBA, spezifiziert von der *Object Management Group* OMG, eine der ersten und daher bekanntesten. Von der Spezifikation ist inzwischen die dritte Version erhältlich. An der grundlegenden Funktionsweise hat sich dabei jedoch nichts verändert; [Eichler] erläutert diese ausgiebig und gut [Eic02]. Das gleiche gilt für das Java-Komponentenmodell Enterprise Java Beans EJB.

Hier soll eine Architektur beschrieben werden, die erst in neuerer Zeit auf den Markt gebracht wurde: Microsofts .NET.

Als weiteres Beispiel wird *SOFA* vorgestellt, eine Komponentenarchitektur, die an der Charles-Universität in Prag entwickelt wird. Sie befindet sich noch in der Entwicklung, eignet sich aber für eine Gegenüberstellung mit dem kommerziell orientierten *.NET*.

Das Ziel der Präsentation beider Systeme besteht darin, die in ihnen umgesetzten Konzepte zu bestimmen und gegebenenfalls zu prüfen, ob sich eines davon für den Einsatz in *Renew* eignet. Daher wird anschließend an die Vorstellung eine Diskussion erfolgen, inwiefern die beiden Architekturen für *Renew* verwendbar sind.

3.4.1 Microsofts *.NET* Framework

Bei dem *.NET* Framework von Microsoft handelt es sich um den Nachfolger früherer Komponentenarchitekturen des gleichen Unternehmens. Diese sollen der Vollständigkeit halber hier noch einmal erwähnt werden:

COM Bei Microsofts *Component Object Model* COM handelt es sich um den ersten komponentenbasierten Ansatz von Microsoft. Dieser entstand aus dem Inter-Application-Protocol *Object Linking and Embedding* OLE, das für den Datenaustausch zwischen Microsoft-Anwendungen wie der Textverarbeitung *MS Word*, der Tabellenkalkulation *MS Excel* und dem Datenbanksystem *MS Access* verwendet wurde.

Anstatt lediglich die Verständigung zwischen zwei Applikationen zu regeln, ist der Ansatz von COM, die Kommunikationspartner selbst in Komponenten zu kapseln. Dadurch ist beispielsweise beim Einfügen einer Tabelle in Word die Excel-Komponente für die Anzeige und Bearbeitung dieses Objektes zuständig. COM ist ein auf rein lokale Verwendung angelegtes Modell. Einer der Vorteile von COM ist die Binärkompatibilität zwischen Objekten verschiedener Sprachen: wenn ein entsprechender Compiler zur Verfügung steht, lässt sich mit jeder beliebigen Sprache ein COM-Objekt erzeugen. Microsoft selbst liefert mit dem *VisualStudio* Unterstützung für C++ und VisualBasic, inzwischen auch für Java.

DCOM Bei DCOM (*Distributed COM*) handelt es sich um die verteilte Version von COM. Entwickelt wurde es von Microsoft im Jahr 1996. In der Microsoft-Terminologie werden DCOM-Objekte als *ActiveX-Controls* bezeichnet.

Die Unterschiede zu COM liegen eben in der Verteiltheit des Ansatzes. Daher beschränken sich die neu eingeführten Elemente auf:

- die Erzeugung entfernter Objekte
- den Zugriff auf entfernte Objekte
- Sicherheitsmechanismen bei Zugriff auf entfernte Objekte.

Als direkter Konkurrent des verteilten Konzeptes von DCOM ist die *Remote Method Invocation* RMI der Sprache Java zu sehen, in der als Komponentenmodell das der *JavaBeans* mitgeliefert werden. Dabei haben die jeweiligen Technologien unter unterschiedlichen Gesichtspunkten Vor- und Nachteile:

Sprache während sich RMI auf die Verwendung von Java beschränkt, handelt es sich bei DCOM um ein sprachunabhängiges Konzept.

Betriebssysteme DCOM läuft lediglich auf Plattformen mit COM-Unterstützung. Damit beschränkt sich die Verwendung effektiv auf Betriebssysteme der MS Windows-Familie. Für RMI wird eine Java Virtual Machine benötigt; diese ist für die wichtigsten Betriebssysteme wie MS Windows, MacOS, Linux und die meisten anderen UNIXe erhältlich.

Geschwindigkeit Da COM und DCOM von den verwendeten Compilern in nativen Maschinencode übersetzt wird, während Java in der Virtual Machine interpretiert wird, ist Java langsamer bei der Ausführung des Codes. Hier ist jedoch zu sagen, dass sich bereits so genannte *Just-in-time*-Compiler durchsetzen, die Java zur Interpretationszeit ebenfalls in nativen Code übersetzen und eine Geschwindigkeitsverbesserung bringen.

Sicherheit Microsofts Sicherheitskonzept für DCOM basiert auf Sicherheitszertifikaten, die von bestimmten Stellen an Entwickler von Komponenten ausgegeben werden. Die Entscheidung, ob eine Komponente als sicher anzusehen ist, liegt dabei letztendlich beim Benutzer, der bestimmen kann, ob er das Sicherheitszertifikat akzeptiert. Tut er dies, so kann DCOM auf dem betreffenden Rechner prinzipiell beliebigen Code ausführen. Java hingegen bringt mit der Java Virtual Machine eine *Sandbox*[1] mit, in der der ausgeführte Code überwacht wird.

.NET Die aktuell von Microsoft vertretene Komponententechnologie wird als .NET bezeichnet. Die hier aufgeführten Erläuterungen stammen aus [Mic03] und [LTQL02]. Microsoft selbst bezeichnet *.NET* als

> Plattform, um Web Services und Applikationen zu entwickeln, auszuliefern und auszuführen ("a *platform for building, deploying and running Web Services and application*")

[1] Mit *Sandbox* wird der Bereich bezeichnet, für den das in Java implementierte Sicherheitsmodell gültig ist. Das Bild bedeutet so viel wie *Sandkasten*, in dem der Code beliebig *spielen* kann, ohne sich auf Speicher außerhalb dieses Bereiches auszuwirken.

Grob gegliedert besteht *.NET* aus folgenden Teilen:

Common Language Runtime Die *CLR* ist das Herzstück von *.NET*. Hierbei handelt es sich um die Einheit, die für die Codeverwaltung zuständig ist, also für das Laden und Ausführen von Komponenten. Vergleichbar ist sie mit der Java Virtual Machine JVM. Sie erfüllt folgende Aufgaben:

- Trennung des Speichers der einzelnen Applikationen
- Verifikation der Typsicherheit
- Übersetzung der IL (intermediate language, s.u.) in nativen Code
- Zugriff auf Metadaten, die zusätzliche Typinformationen enthalten
- Speichermanagement für verwaltete Objekte
- Sicherheitsüberprüfung bei Codezugriff
- Ausnahmebehandlung, auch über Sprachgrenzen hinweg
- Unterstützung der Zusammenarbeit zwischen verwaltetem Code, COM-Objekten und anderem Code
- Funktionen zur Unterstützung von Entwicklern wie Hilfe bei Geschwindigkeitsoptimierung und Fehlersuche

Common Type System Das *CTS* ist das Typsystem von *.NET*. Es bietet eine einheitliche Typisierung von Objekten sowie eine Anzahl von Operationen, wie sie in vielen Programmiersprachen gebräuchlich ist.

Common Language Specification Die *CLS* enthält Spezifikationen, welche Art von Konstrukten erlaubt sind oder benötigt werden, um Code auf der CLR ausführen zu können. Es handelt sich dabei um eine Untermenge des Common Type Systems. Durch die CLS wird die Typsicherheit über verschiedene Programmiersprachen hinweg erreicht, eines der erklärten Ziele von *.NET*.

Microsoft Intermediate Language Die MSIL, oder nur IL, ist eine Zwischensprache, in die *.NET*-Code übersetzt wird. Es handelt sich um die Spezifikation einer *virtuellen* Sprache, vergleichbar mit dem Bytecode von Java. Die Überführung eines Programms in die MSIL wird ebenfalls durchgeführt, um die Sprachunabhängigkeit von *.NET* umzusetzen. Anders als der Java-Bytecode wird die MSIL jedoch vor ihrer Ausführung in nativen Code übersetzt, um keinen Geschwindigkeitsverlust zu erleiden.

Assembly Eine *Assembly* ist bei *.NET* die Entsprechung einer Komponente. Microsoft selbst nennt sie auf der *.NET*-Hompage [Mic03] eine

> Sammlung von Funktionalität, die als eine einzelne Implementationseinheit erstellt, versioniert und verteilt wird ("*It is a collection of functionality that is built, versioned, and deployed as a single implementation unit*");

hierbei handelt es sich im Allgemeinen um eine eigenständige Applikation. Sie kann jedoch auch im Komponentensinn als Dienstanbieter für andere *Assemblies* verwendet werden.

Zu einer *Assembly* gehören Metainformationen, die sie beschreiben. Diese Beschreibung nennt sich *Manifest* und enthält

- die Identität der *Assembly* in Form eines textuellen Namens
- eine digitale Signatur für Sicherheitsprüfungen
- eine Liste von Dateien, die zu der *Assembly* gehören
- eine Spezifikation der Typen und Ressourcen, die die *Assembly* zur Verfügung stellt
- eine Liste der Abhängigkeiten der *Assembly*
- eine Angabe von Sicherheitseinstellungen, die zur Ausführung der *Assembly* benötigt werden.

Dieses Manifest wird von der CLR verwendet, um Referenzen aufzulösen, Versionsabhängigkeiten sicherzustellen und eine Überprüfung der Korrektheit der *Assembly* durchzuführen. Die enthaltenen Informationen sind sehr umfangreich, können aber zu großen Teilen automatisch erstellt werden. Dies geschieht im Allgemeinen durch von Microsoft ausgelieferte Werkzeuge zum Erstellen von *.NET*-Assemblies. Umgekehrt bedeutet dies jedoch, dass sich *.NET*-Anwendungen und -Komponenten kaum ohne diese Werkzeuge erstellen lassen, da der Mehraufwand, die Metainformationen in das Manifest einzutragen, schon bei kleinen Programmen nicht vertretbar ist.

Ein weiterer Nachteil von *.NET* ist, dass die Common Language Runtime, also der Kernbestandteil dieses Systems, nur für Microsoft- Betriebssysteme erhältlich ist. Damit beschränkt sich die Verwendung von *.NET*-Produkten auf Computer mit MS Windows.

Dem gegenüber steht die Sprachunabhängigkeit durch Verwendung der Common Language Specification. Außerdem ist *.NET* nicht auf eine bestimmte Hardwarearchitektur angewiesen: durch die Verwendung der Intermediate Language, die erst bei Benutzung einer *Assembly* auf dem Zielsystem in nativen Code übersetzt wird, ist es unwichtig, auf welchem Prozessor das Programm laufen soll - so lange Windows installiert ist, da nur hierfür eine CLR zur Verfügung steht.

3.4.2 *SOFA*

Bei dem Projekt SOFtware Appliances *SOFA* der Charles-Universität in Prag handelt es sich um eine Komponentenplattform für die Softwareentwicklung mit Komponenten. Die hier zusammengetragene Information sind der Homepage [Dis03] entnommen.

Dabei werden Applikationen als Hierarchien von zusammengesetzten Komponenten gesehen, die als *primitiv* oder *zusammengesetzt* betrachtet werden, je nachdem, ob sie weitere Komponenten enthalten.

Komponenten

Eine Komponente wird durch einen *Rahmen* (*frame*) sowie seine *Architektur* (*architecture*) beschrieben. Der Begriff des Rahmens entspricht in etwa dem der Schnittstelle in der objektorientierten Programmierung: es handelt sich um eine black-box-Sicht auf die Komponente. Dabei wird festgelegt, welche Dienste sie anbietet (*provides*) und auf welche sie zurückgreift (*requires*), wodurch die in Abschnitt 3.1 definierte Abhängigkeit zwischen Komponenten fest gelegt wird. Außerdem enthält der Rahmen die Beschreibung, wie die Komponente parametrisiert werden kann.

Die *Architektur* beschreibt, wie eine Komponente ihre Aufgabe erfüllt: dabei wird angegeben, welche Komponenten zu instantiieren sind sowie auf welche Art und Weise diese miteinander verbunden sind.

Von diesen Verbindungsbeschreibungen gibt es vier Arten:

Binden bedeutet, dass die *requires*-Schnittstelle mit der *provides*-Schnittstelle der anderen Komponente verbunden ist

Delegation bedeutet, dass eine Komponente ihre *provides*-Schnittstelle durch Zugriff auf die *provides*-Schnittstelle einer anderen Komponente erfüllt

Subsumierung bedeutet, dass die *requires*-Schnittstellen zweier Komponenten miteinander verbunden sind

Befreiung bedeutet, dass keine Verbindungen vorhanden sind.

Konnektoren

Verbindungen werden durch *Konnektoren* realisiert. Diese sind wie Komponenten auch First-Class Objekte, befinden sich also konzeptionell auf der gleichen Ebene wie Komponenten. Aus diesem Grunde werden zur Spezifikation von Konnektoren ebenfalls *Rahmen* und *Architektur* als Elemente zur Beschreibung verwendet.

SOFA stellt drei vordefinierte Typen von Konnektoren zur Verfügung:

- Prozeduraufrufe
- Events und
- Datenströme

außerdem können neue Arten von Konnektoren definiert werden.

Verhaltensprotokolle

Die Kommunikation zwischen Komponenten wird bei *SOFA* durch *Verhaltensprotokolle* formal beschrieben. Diese Beschreibung gehört zu der Spezifikation des Rahmens der Komponente: zusätzlich zu den zur Verfügung stehenden Methoden gibt der Entwickler eine Reihenfolge an, in der sie aufgerufen werden können. Dabei wird eine Notation ähnlich der regulärer Ausdrücke verwendet.

Insgesamt ist *SOFA* ein interessanter Ansatz, zumal alle Bestandteile in einer einheitlichen, formal verifizierbaren Form vorliegen. Das Projekt befindet sich jedoch noch in einer Entwicklungsphase.

3.4.3 Gegenüberstellung

Nachdem die existierenden Komponentenmodelle vorgestellt worden sind, sollen die jeweils umgesetzten Konzepte miteinander verglichen und anschließend überprüft werden, ob sich eins davon für den Einsatz in *Renew* eignet.

Vergleich

Die beiden Systeme, *.NET* und *SOFA*, unterscheiden sich sowohl in Umfang als auch darin, zu welchem Zweck sie entwickelt wurden.

Bei *SOFA* handelt es sich um ein akademisches Projekt, das sich in der Entwicklung befindet. Anwendung findet es hauptsächlich in der Forschung zum Verhalten dynamischer Komponenten [MP00]. Ein großer Vorteil dieses Systems ist es, dass das Verhalten der Komponenten formal spezifiziert werden kann [PV02].

Die *.NET*-Architektur hingegen wird bereits von verschiedenen Herstellern eingesetzt. Dies liegt einerseits daran, dass Microsoft seine Marktmacht stark für die Verbreitung von *.NET* eingesetzt hat. Andererseits ist *.NET* aber auch ein durchdachtes System und ausgereifter als *SOFA*.

Zudem besteht der Zweck von *.NET* darin, eine einheitliche Entwicklung für die Betriebssysteme von Microsoft bereit zu stellen. Dafür ist die Verwendung der CLR von Nöten, die zugleich ebenfalls Eigenschaften wie Unabhängigkeit von der eingesetzten Programmiersprache ermöglicht. Zusätzlich tragen die *Assemblies* dazu bei, die enthaltene Software einfacher ausliefern zu können, indem sie alle benötigten Codebibliotheken mitliefern kann, wodurch Kompatibilitätsprobleme vermieden werden. Für Entwickler, die nur Microsoft-Produkte verwenden, ist *.NET* daher ein sinnvoller Schritt.

Einsatz in *Renew*

Eignet sich nun eines der vorgestellten Systeme dazu, die in *Renew* angestrebten Komponenten zu verwalten?

Zu *SOFA* ist zu sagen, dass es zwei *SOFA*-Implementationen gibt: den *SOFA*-Prototyp [sPP03] und eine experimentelle Implementation für C++ [Kal01]. *Renew* hingegen ist über ein Prototyp-Stadium hinaus, und die Einführung von Komponenten soll die Stabilität des Systems nicht gefärden. Daher kommt *SOFA* nicht für diese Arbeit in Frage.

Auch *.NET* ist für den Einsatz mit *Renew* nicht geeignet, da sich der Einsatz von *.NET* auf die verschiedenen Windows-Varianten beschränkt. *Renew* hingegen muss unter dem Betriebssystem Solaris vergewendet werden können, da die Universität Hamburg dieses nutzt; hier wird viel der Entwicklungsarbeit an *Renew* geleistet.

Kapitel 4

Plugins

Das vorige Kapitel beschreibt das Konzept der Komponentenorientierung. Komponenten sind Softwareeinheiten, die fachliche Domänen abdecken; sie werden von einer Verwaltungsinstanz geladen, die auch die Kommunikation zwischen ihnen regelt. Diese Kommunikation ist die einzige Möglichkeit, wie eine Komponente die Funktionalität einer anderen verwenden kann.

Dieses Kapitel führt eine spezielle Form der Komponente ein, das Plugin; der Zweck eines Plugins ist es, das Verhalten einer anderen Komponente zu verändern. Dies dient dazu, die in der Einleitung geforderte *Erweiterbarkeit* umzusetzen.

Als erstes wird der Begriff *Plugin* definiert; anschließend werden zwei Pluginsysteme und ihre Konzepte vorgestellt und verglichen. Im nächsten Kapitel wird mit Hilfe von Referenznetzen eine Visualisierung der Konzepte *Komponenten* und *Plugins* vorgestellt.

4.1 Begriffe

Die Begriffe *Plugin* oder *plugin-fähig* werden im Zusammenhang mit vielen aktuellen, populären Softwareprodukten verwendet. Dabei fehlt eine echte Definition von *Plugin*. Statt dessen verlassen sich die Hersteller darauf, dass das eingängige Bild des *Hereinsteckens* von Software in ein bestehendes System eine genaue Begriffsbestimmung überflüssig macht.

Eichler vertritt in [Eic02] die Auffassung, ein Pluginsystem sei ein Komponentensystem, das die Verwendung dynamischer Komponenten erlaubt. Die Eigenschaft der Dynamik soll jedoch in der vorliegenden Arbeit nicht der entscheidende Faktor sein, der ein Plugin ausmacht. Zudem ist bereits in Abschnitt 5.2 von [Eic02] ein Gegenbeispiel zu finden: bei GIMP, dem GNU Image Manipulation Program, werden Plugins zur Startzeit des Programmes geladen und können später nicht hinzugefügt werden; dies ist jedoch gerade die Definition dynamischer Komponenten. Ebenso verhält es sich mit dem Pluginsystem *eclipse*: auch dieses lädt Komponenten lediglich beim Starten. Mit *equinox* [equ03] wird zur Zeit zwar an einem Komponentenmodell gearbeitet, das dynamische Komponenten erlauben soll, sich jedoch noch in der Entwicklung befindet. Dennoch kann man *eclipse* schon in der vorliegenden Form als *Pluginsystem* bezeichnen.

Dabei spielt es keine Rolle, ob Plugins im intuitiven Sinne oder in dem der folgenden Definition verstanden werden. Bei *eclipse* handelt es sich sogar um nichts anderes als um eine Plugin-Verwaltung, die sämtliche fachliche Funktionalität durch Plugins zur Verfügung stellt. Eine genauere Beschreibung des von *eclipse* gebotenen Plugin-Konzepts findet sich in Abschnitt 4.2.2.

Eine alternative Definition von *Plugin* bietet [MV03]:

> *Plugins* sind Ergänzungen zu einer Applikation, die von dieser vorgesehen sind, die sie aber nicht selbst mitbringen kann oder will. Ein Plugin deckt dabei eine fachliche Funktion so vollständig ab, dass keine andere Komponente des Systems etwas Spezifisches darüber wissen müsste. Für die dazu nötigen Schnittstellen ist die Applikation verantwortlich.

Hier werden Plugins also als Ergänzungen gesehen, die ihre Funktionalität in ein bestehendes System einbringen, das eigens dazu Schnittstellen zur Verfügung stellt.

In der Begriffsbestimmung, die für diese Arbeit maßgeblich ist, werden die beiden aufgeführten Definitionen zusammengeführt. Dabei werden Plugins als Spezialfall von Komponenten betrachtet. Aber wann ist eine Komponente ein Plugin?

Die zweite Definition besagt, dass ein Plugin die Funktionalität des Systems erweitert, dies geschieht jedoch durch jede Komponente. Ein Plugin im Sinne dieser Arbeit ergänzt aber nicht nur das Gesamtsystem, sondern beeinflusst das Verhalten bereits vorhandener Komponenten, die die dafür notwendige Schnittstelle zur Verfügung stellen.

Plugin Plugins sind Komponenten, die das Verhalten einer oder mehrerer anderer Komponenten im System verändern. Dies geschieht über von diesen zur Verfügung gestellte Schnittstellen.

Für die Abhängigkeit zwischen den Komponenten bedeutet dies, dass das Plugin immer von der Komponente abhängig ist, die es erweitert, da diese die Beschreibung der Schnittstelle anbieten muss, über die die Erweiterung stattfindet.

Weiterhin ist anzumerken, dass der Begriff des Plugins anders als in [Eic02] nicht dynamische Komponenten bezeichnet. Dies sind vielmehr zueinander orthogonale Konzepte: ein System kann erlauben, dynamisch Komponenten hinzuzuladen, die sich jedoch

nicht gegenseitig erweitern können. Andererseits besteht die Möglichkeit, dass ein Pluginsystem nicht erlaubt, zu den beim Start vorhandenen Komponenten neue hinzuzufügen.

Plugins und Komponenten

Wie aus der Definition ersichtlich wird, handelt es sich bei Plugins um Komponenten. Dies bedeutet, dass sie von einem Komponentensystem geladen und verwaltet werden.

Der Unterschied zu "reinen" Komponenten besteht darin, in welcher Umgebung sie sich befinden: während sich Komponenten auf einer Plattform aufhalten, sind Plugins zusätzlich in anderen Komponenten bekannt, nämlich in denen, die sie erweitern (dies wird in Kapitel 5 veranschaulicht). Daher kann eine Komponente, die durch ein Plugin erweitert wird, mit diesem kommunizieren, ohne den Kommunikationsdienst der Plattform in Anspruch zu nehmen.

Eine weitere Anmerkung betrifft den Punkt der *fachlichen Domäne*, die eine Komponente abdeckt: damit ein Plugin das Verhalten einer vorhandenen Komponente sinnvoll beeinflussen kann, muss es teilweise in dem fachlichen Bereich arbeiten, für den die erweiterte Komponente zuständig ist. In der hier vertretenen Sicht bietet eine Komponente also *grundlegende* Funktionalität für die Behandlung einer fachlichen Domäne, die von Plugins um *spezialisierte* ergänzt wird.

4.2 Existierende Pluginsysteme

In diesem Abschnitt werden einige Pluginsysteme vorgestellt. Dies dient dazu, grundlegende Strukturen zu finden, die in das spätere für *Renew* zu entwickelnde System einfließen sollen.

Wie schon für Abschnitt 3.4 gilt auch hier der Verweis auf [Eic02], wo sich eine Auflistung verschiedener Pluginsysteme findet (hier werden wieder nur solche Systeme beschrieben, die dort nicht beschrieben sind).

Insbesondere wird darauf Acht gegeben, wie in den bestehenden Systemen die Erweiterbarkeit umgesetzt wird, da sich dies in Abschnitt 3.3 als Schwäche der reinen Komponentenorientierung erwiesen hat.

4.2.1 GStreamer

Bei *GStreamer* handelt es sich um ein Framework zur Erzeugung von *Streaming Media*-Applikationen. Es verfügt über einen Plugin-Mechanismus, mit dem Entwickler neue Funktionalität hinzufügen können.

Elemente

Konzeptionell besteht *GStreamer* aus so genannten *Elementen*. Dabei kann es sich hauptsächlich um eine von drei Arten handeln: *Source*(Quell)-Elemente, die einen Datenstrom erzeugen, den *Sinks* (Senken) konsumieren. Dazwischen können *Filter*-Elemente gehängt werden, die den Datenstrom bearbeiten. Technisch gesehen sind Elemente Klassen, die von `GstElement` abgeleitet sind.

Weitere Elementarten sind *Autoplugger*, die dafür verantwortlich sind, verschiedene Elemente miteinander zu verknüpfen, sowie *Bins*, die eine Kollektion von Elementen repräsentieren. Letztere werden häufig als Scheduler der zugehörigen Elemente eingesetzt, um einen ungestörten Datenfluss zu gewährleisten.

Um neue Elemente in das System einbringen zu können, muss ein Entwickler diese in einem Plugin verpacken. Der Unterschied zwischen Elementen und Plugins ist der, dass ein Element die Funktionalität bietet, das Plugin[1] hingegen zur Verwaltung dient, was zu der Komponentendefinition aus Abschnitt 3.1 passt. Ein Plugin kann mehrere Elemente zur Verfügung stellen.

Pads und Buffer

Einen weiteren wichtigen Bestandteil des *GStreamer*-Ansatzes sind die so genannten *Pads*. Diese repräsentieren Verknüpfungspunkte, durch die die einzelnen Elemente miteinander verbunden werden. Im Gegensatz zu den Autopluggern handelt es sich bei diesen jedoch nicht um Elemente, sondern vielmehr um eine Spezifikation, welche Art von Daten aus einem Element ein- respektive ausgehen kann.

Die Datenströme selbst werden über so genannte *Buffer* übertragen. Dies sind Strukturen, die den aufgeteilten Strom enthalten und beschreiben. Zu diesen Metadaten gehören unter anderem Informationen, wo der Buffer liegt, wie lang er ist, sowie ein Referenzzähler. Dieser führt Buch über die Elemente, die sich auf diesen Buffer beziehen. Er gibt den vom Buffer benötigten Speicherplatz wieder frei, wenn er nicht mehr gebraucht wird und sorgt dafür, dass dies nicht zu früh geschieht.

[1] Die Verwaltungsobjekte sind Exemplare der Klasse *GstPlugin*

Bewertung

Eichler sagt in [Eic02, S. 85f] über Netscape-Plugins, sie seien "im wesentlichen darauf [ge]zielt, besondere Inhalte einer Webseite anzeigen zu können". In *GStreamer* sind Plugins dafür konzipiert, mit einem Strom von Multimediadaten umzugehen. Diese Anwendung spielt beim Entwurf eines Pluginsystem für *Renew* keine Bedeutung, das Konzept ist jedoch interessant.

Insbesondere treten bei der Kommunikation zwischen Komponenten, wie sie in *Renew* vorkommen, keine großen Datenmengen auf. Daher wird keine Speicherverwaltung benötigt, wie die *Buffer* sie zur Verfügung stellen. Konzeptionell interessant ist an den *Buffern* jedoch die Tatsache, dass sie Metainformationen mit sich tragen. Dies entspricht etwa typisierten Nachrichtenobjekten, die zwischen Komponenten versendet werden.

In Zusammenhang damit sind besonders die *Pads* von Bedeutung. Diese entscheiden, ob gewisse Buffer von den ihnen zugehörigen Elementen versendet bzw. empfangen werden können. Dies eröffnet Möglichkeiten, die vorhandenen Elemente mit ihren Pads graphisch darzustellen und miteinander zu verbinden. Dabei kann die Repräsentation der Pads gestalterisch so gewählt werden, dass ersichtlich wird, welche Elemente miteinander verbunden werden können. Eine ähnliche Möglichkeit für die komponentenorientierte Programmierung ist etwa auf Seite 137 von [Gri98] dargestellt.

Insgesamt handelt es sich bei dem *GStreamer*-Pluginsystem um einen interessanten Ansatz. Die niedrige Versionsnummer - zur Zeit des Schreibens dieser Arbeit[2] 0.6.2 - deutet aber darauf hin, dass sich noch viel an dieser Open-Source-Software verändern wird.

4.2.2 *eclipse*

Bei dem *eclipse*-Projekt handelt es sich um eine Plattform zur Entwicklung von Integrated Development Environments (IDEs) [ecl03]. Obwohl *eclipse* selbst vollständig in Java programmiert ist, ist es nicht auf die Verwendung mit Java oder einer anderen bestimmten Programmiersprache festgelegt, sondern überlässt die Implementation der Unterstützung einer spezifischen Sprache anderen Herstellern.

Das Interessante an diesem Konzept im Zusammenhang mit dieser Arbeit ist, dass die Werkzeuge, die diese Unterstützung liefern, vollständig über Plugins eingebunden werden. *eclipse* bietet dabei lediglich einige allgemeine Funktionen, die die Zusammenarbeit der Komponenten unterstützen, sowie einen Plugin-Mechanismus zu ihrer Verwaltung.

Auch in *eclipse* wird ein Plugin als Komponente begriffen, die einen bestimmten Dienst zur Verfügung stellt. Plugins sind in diesem Zusammenhang nicht als dynamische Komponenten zu sehen, sondern müssen beim Start von *eclipse* zur Verfügung stehen. Jedes Plugin ist als Exemplar einer Plugin-Klasse vorhanden, die für die Konfiguration und die Steuerung des Plugins sorgt. Die Klasse *org.eclipse.core.runtime.Plugin*, von der alle Plugin-Klassen abgeleitet sein müssen, legt diese Funktionalität fest.

Zwei Methoden dieser Klasse verdienen besondere Erwähnung: **startup** und **shutdown**. Diese werden vom Plugin-Management verwendet, um den Lebenszyklus des Plugin zu steuern. Sie werden aufgerufen, bevor ein Plugin das erste Mal respektive nicht mehr verwendet wird. Dadurch kann ein Plugin eventuell benötigte Ressourcen beim Start laden und diese beim Beenden wieder freigeben.

Bis hierher entspricht das Konzept eher dem Begriff der Komponenten. Wie unten beschrieben, stellt *eclipse* jedoch einen Mechanismus bereit, mit dem sich Komponenten gegenseitig erweitern und damit zu Plugins werden.

[2] Im August 2003

Metainformation

Zusätzlich zu dem ausführbaren Code enthält ein Plugin Metainformationen, mit denen *eclipse* das Plugin zu verwaltet. Dazu trägt die Verwaltung die relevanten Daten in eine Plugin-Registry ein. Die Informationen liegen als *Manifest* genannte XML-Datei vor und enthalten mindestens:

- den Namen des Plugins
- einen eindeutigen String zur Identifikation
- eine Versionsnummer
- Information, wo sich der Code des Plugins befindet (normalerweise eine oder mehrere jar-Dateien)
- der Hersteller des Plugins

Dies ist eine Plugin-Beschreibung, in der die Minimalinformation enthalten ist:

```
<?xml version="1.0" encoding="UTF-8"?>
<plugin
  name="Plugin Name"
  id="de.plugin.name"
  version="1.0"
  provider-name="MyProvider.de">
  <runtime>
    <library name="codelocation.jar">
      <export name="*"/>
    </library>
  </runtime>
</plugin>
```

Einsatz eines Plugins

Die Auslieferung eines *eclipse*-Plugins geschieht, indem für das Plugin ein Ordner unterhalb des Ordners **plugins** im *eclipse*-Verzeichnis erstellt wird. In diesen werden die XML-Datei mit den Metainformationen, jar-Dateien mit dem Java-Code und andere benötigte Resourcen hineinkopiert. Dort findet die Plugin-Verwaltung die Informationen und erzeugt einen Eintrag in der Plugin-Registrierung. Wenn das Plugin dann benötigt wird, wird es geladen und kann verwendet werden. Das Laden der Klasse und Erzeugen eines Exemplars des Plugins heisst in der *eclipse*-Nomenklatur *Aktivierung* (*Activation*).

Erweiterung von Plugins

Wie erwähnt ist in diesem Kapitel die Erweiterbarkeit zwischen Plugins von besonderer Bedeutung. Wie ist nun in *eclipse* das Konzept realisiert, eine Komponente durch eine andere, ihr unbekannte, erweitern zu lassen?

Dazu ist es zunächst von Bedeutung, weitere mögliche Einträge in der XML-Datei mit den Metainformationen zu erläutern. Diese beziehen sich auf die *Abhängigkeit* zwischen zwei Plugins.

Abhängigkeit Dabei wird zum einen ein ähnliches Konzept verfolgt, wie schon das in Abschnitt 3.4.2 beschriebene Komponentenmodell SOFA: Plugins, die das Plugin für seine Funktionsfähigkeit benötigt, werden in ein *requires*-Feld eingetragen. Diese Abhängigkeit bezieht sich sowohl auf die Compile- als auch auf die Laufzeit: das Plugin kann weder kompiliert werden, ohne dass das entsprechende andere Plugin als Bibliothek vorhanden ist, noch kann es initialisiert und ausgeführt werden. Letzteres wird von der Plugin-Verwaltung von *eclipse* sichergestellt. Ein solcher Eintrag würde folgendermaßen aussehen:

```
<?xml version="1.0" encoding="UTF-8"?>
<plugin
   id="de.abhaengig"
   name="Abhaengiges Plugin"
   version="1.0.0">
   <runtime>
      <library name="code.jar"/>
   </runtime>
   <requires>
      <import plugin="de.abhaengigkeit"/>
   </requires>
</plugin>
```

Erweiterung Andererseits gibt es zusätzlich die Möglichkeit, ein anderes Plugin mit Funktionalität anzureichern. Dies muss auf beiden Seiten, also sowohl bei dem erweiternden als auch bei dem erweiterten, vorgesehen werden.

Zu erweiterndes Plugin Bei diesem kann es vorkommen, dass es mehrere Stellen vorsieht, an denen eine Erweiterung möglich sein soll. So erlaubt die graphische Oberfläche es beispielsweise, sowohl zusätzliche Menupunkte hinzuzufügen als auch neue Editoren zur Bearbeitung eines bestimmten Dateityps anzumelden. Diese voneinander unterscheidbaren Stellen werden im Plugin-Manifest getrennt angegeben. Sie nennen sich *Extension Point*s, also etwa *Erweiterungspunkte*.

Diese *Extension Point*s werden beim zu erweiternden Plugin als Eintrag in das Manifest aufgenommen (Beispiel aus [Bol03]):

```
<?xml version="1.0" encoding="UTF-8"?>
<plugin
 id="org.eclipse.ui"
    name="Eclipse UI"
    version="2.1.0"
    provider-name="Eclipse.org"
    class="org.eclipse.ui.internal.UIPlugin">
 <extension-point id="actionSets" name="Action Sets"
      schema="schema/actionSets.exsd"/>
</plugin>
```

Das Beispiel zeigt einen Ausschnitt aus dem Eclipse UI Plugin, das die Möglichkeit bietet, Menüs, Menüeinträge und Toolbuttons als *ActionSet*-Erweiterung anzumelden. Eine bisher noch nicht erwähnte Angabe ist der Parameter *schema* des

*Extension Point*s. In dieser wird ein XML-Schema angegeben, die das erweiternde Plugin in seinem Manifest verwenden muss, um für die Erfüllung des *Extension Point*s notwendige Information zur Verfügung zu stellen. Hierfür gibt es für jeden *Extension Point* eine Referenzseite, in der eine genaue Dokumentation verfügbar ist, welche Einträge in diesem Schema was bedeuten.

Erweiterndes Plugin Das erweiternde Plugin muss in seinem Manifest angeben, welche *Extension Point*s es erweitert und die dazu erforderlichen Informationen mitliefern. Da es möglich ist, dass mehrere Plugins *Extension Point*s mit dem gleichen Namen definieren, wird die eindeutige ID des korrekten Plugins als Präfix verwendet.

```
<plugin
        id="org.eclipse.help.ui"
        name="Help System UI"
        version="2.1.0"
        provider-name="Eclipse.org"
        class=
          "org.eclipse.help.ui.internal.WorkbenchHelpPlugin">
    <!-- ... -->
    <!-- Action Sets -->
    <extension
(1)      point="org.eclipse.ui.actionSets">
(2)     <actionSet
               label="Help"
               visible="true"
               id=
                "org.eclipse.help.internal.ui.HelpActionSet">
(3)      <action
               label="&Help Contents"
               icon="icons/view.gif"
               helpContextId=
                "org.eclipse.help.ui.helpContentsMenu"
               tooltip="Open Help Contents"
(4)             class=
                "org.eclipse.help.ui.internal.
                 HelpContentsAction"
               menubarPath="help/helpEnd"
               id="org.eclipse.help.internal.ui.HelpAction">
           </action>
           <!-- ... other actionSet elements -->
(3)      <action
               label="&Help..."
               icon="icons/search_menu.gif"
               helpContextId=
                "org.eclipse.help.ui.helpSearchMenu"
               class=
                "org.eclipse.help.ui.internal.
                 OpenHelpSearchPageAction"
               menubarPath=
```

```
                "org.eclipse.search.menu/dialogGroup"
               id="org.eclipse.help.ui.OpenHelpSearchPage">
            </action>
        </actionSet>
    </extension>
<!-- ... -->
</plugin>
```

In dem Beispielmanifest wird der oben spezifizierte *Extension Point actionSets* angegeben. Als Präfix wird, wie an Stelle (1) zu sehen, *org.eclipse.ui* verwendet, also der *id*-Eintrag aus dem vorherigen Manifest.

Wie aus der Referenzseite für den *Extension Point org.eclipse.ui.actionSets* ersichtlich ist, besteht ein actionSet aus einem Header (2) sowie einer Anzahl von *action*-Einträgen (3). Diese wiederum enthalten Informationen, an welchem Punkt sie in das Menü einzufügen sind, welche Anschriften an den Menüeinträgen stehen, welche Bilddatei zur Darstellung eines Toolbuttons verwendet werden soll und so fort.

Insbesondere ist noch der Eintrag *class* wichtig (4). Dieser gibt an, welche Java-Klasse dafür zuständig ist, die Auswahl des entsprechenden Menüpunktes oder Toolbuttons zu behandeln. Zu dieser Klasse findet sich in der Dokumentation des *Extension Point*s `org.eclipse.ui.actionSets`, dass sie eines der Interfaces `org.eclipse.ui.IWorkbenchWindowActionDelegate` oder dessen Subinterface `org.eclipse.ui.IWorkbenchWindowPulldownDelegate` implementieren muss. In diesen findet sich die Methode `public void run(org.eclipse.jface.action.IAction action)`, die vom erweiterten Plugin aufgerufen wird, wenn der entsprechende Menüpunkt gewählt wurde.

Bewertung von *eclipse*

Die Entwicklungsplattform *eclipse* verfügt über Mechanismen, mit denen Plugins geladen und verwaltet werden können. Diese Plugins stellen die gesamte Endbenutzerfunktionalität zur Verfügung. Bei *eclipse*-Plugins handelt es sich um statische Komponenten, die lediglich beim Start des Systems gesucht und registriert werden.

Plugins können sich gegenseitig um Funktionalität erweitern. Das dabei verwendete Konzept in *eclipse* ist das der *Extension Point*s. Diese werden vom zu erweiternden Plugin definiert. Zur Beschreibung eines *Extension Point*s gehören eine XML-Schema-Datei, in der die benötigten Informationen, die eine Erweiterung zur Verfügung stellen muss, beschrieben werden; ein Vertrag, der die Verpflichtungen und Rechte zwischen Host- und erweiterndem Plugin regelt; sowie eine Dokumentation, in der für den Entwickler der Erweiterung beides erläutert wird.

Im Normalfall handelt es sich bei den Verpflichtungen des Host-Plugins um eine Zusage, unter bestimmten Bedingungen Methoden in einem von der Erweiterung zur Verfügung gestellten Callback-Objekt aufzurufen. Das erweiternde Plugin erfüllt den Vertrag, indem es die benötigte Information aus dem entsprechenden XML-Schema zur Verfügung stellt und diese Einträge im Manifest mit Werten versieht, die laut Dokumentation gültig sind. Dabei ist insbesondere der Typ der angegebenen Callback-Klasse zu berücksichtigen.

Das Pluginsystem, das *eclipse* bietet, ist sehr flexibel und mächtig. Insbesondere ist die Art interessant, wie die Informationen, die für die Zusammenarbeit zwischen

erweiterbarem und zu erweiterndem Plugin notwendig sind, mittels der jeweiligen Manifeste übertragen werden. Zu bemerken ist in diesem Zusammenhang, dass trotz einer hohen Formalisierung dieser Kommunikation die menschenlesbare Dokumentation eine wichtige Rolle spielt.

Nachteile In der Tat ist *eclipse* ein modernes Pluginsystem, das eine eingängige und einheitliche Art und Weise bereit stellt, Plugins zu entwickeln. *eclipse* ist auf die Verwendung als Entwicklungsumgebung ausgelegt und bietet daher viele in diesem Zusammenhang benötigte Grundfunktionen wie die Verwaltung von Projekten, Ordnern und Dateien. Dennoch gibt es auch einige Punkte, die gegen *eclipse* sprechen.

Keine dynamischen Komponenten Nachteilig an der Plugin-Architektur von *eclipse* ist, dass lediglich statische Komponenten als Plugins erlaubt sind. Im Zusammenhang mit dem *equinox*-Projekt, das an verschiedenen technologischen Aspekten des *eclipse*-Plugin-Systems arbeitet, entsteht derzeit ein Modell für dynamische Plugins. Dieses Projekt befindet sich jedoch derzeit im Prototyp-Stadium und hat noch nicht Einzug in die *eclipse*-Plattform gehalten.

Erhöhter Entwicklungsaufwand Der Aufwand, ein Plugin zu schreiben, ist relativ groß. Insbesondere bei kleineren Erweiterungen nimmt der Overhead, den die Einarbeitung in die Dokumentation sowie die Erzeugung des Manifests mit dem etwas sperrigen XML-Format mit sich bringen, viel Zeit in Anspruch. Hierzu muss jedoch gesagt werden, dass sich innerhalb von *eclipse* die Plugin-Entwicklungsumgebung *Plugin Development Environment PDE* gebildet hat, die diesen Nachteil durch Werkzeugunterstützung mindern soll.

Neues Window-Toolkit Bei *eclipse* wird nicht das von Java vorgesehene *Abstract Windowing Toolkit AWT* oder das modernere Toolkit *Swing* verwendet. Vielmehr basiert die GUI hier auf der Neuentwicklung *Standard Widget Toolkit SWT*. Dieses besitzt mehr graphische Elemente als das *AWT*, das nur diejenigen anbietet, die auf allen Plattformen zur Verfügung werden. Zudem ist das SWT schneller als Swing, da dieses die neuen Elemente wie etwa Bäume und Tabellen zwar kennt, jedoch immer emuliert, auch wenn das Fenstersystem diese nativ anbietet. Das SWT emuliert nur dann, wenn das darzustellende Element auf der gegenwärtigen Plattform nicht zur Verfügung steht.

Dies hat zwar Vorteile, bedeutet jedoch, dass Entwickler das neue Framework lernen müssen. Da es recht komplex ist, ist der Aufwand dafür nicht unbeträchtlich. Zudem haben die meisten Java-Programmierer bereits mit dem AWT und/oder Swing Erfahrungen gesammelt, die bei dem SWT nicht mehr anwendbar sind.

Laufzeitverhalten Der letzte Nachteil des *eclipse*-Systems ist sein Laufzeitverhalten. Die aufwändige Verwaltung der Plugins ist sowohl speicher- als auch geschwindigkeitsintensiv. So dauert der Start von *eclipse* mit 75 Plugins an einer SunBlade 1100 mit 256 MB Speicher etwa 30 Sekunden. Der Prozess ist inklusive Daten und Stack 105 MB groß. Auch wenn dies bei modernen Computern kein Problem darstellen sollte, sind dies doch recht hohe Anforderungen an die Rechnerumgebung.

4.2.3 Gegenüberstellung

Die in diesem Kapitel beschriebenen Beispiele zeigen trotz verschiedener Technologie Gemeinsamkeiten in der Struktur, wie sie Plugins behandeln. Diese werden in diesem Abschnitt zusammengefasst, damit sie in den späteren Entwurf eines Plugin- Mechanismus für *Renew* einfließen können. Andererseits sind durch die unterschiedliche Ausrichtung der Systeme auch Unterschiede vorhanden, die ebenfalls hier diskutiert werden.

Als erste Gemeinsamkeit wäre dabei die n:m-Beziehung zwischen Plugin und Erweiterung zu nennen: *eclipse* ermöglicht es, das ein Plugin mehrere *Extension Point*s definiert und dadurch auf verschiedene Art und Weise erweiterbar ist. Andererseits kann ein Plugin ebenfalls auf mehr als nur einen *Extension Point* aufsetzen und somit verschieden andere Komponenten erweitern. Im Falle von GStreamer ist dies dadurch gegeben, dass ein Plugin verschiedene Pads anbieten kann, an die sich die jeweiligen anderen Komponenten anschließen lassen.

Auch bei den Schnittstellen der das Plugin repräsentierenden Klasse zeigen sich Ähnlichkeiten:

- In beiden Schnittstellen gibt es Methoden zur Idenfikation des Plugins im System. Bei GStreamer sind dies etwa `gst_plugin_get_name` und `gst_plugin_get_longname` aus GstPlugin, während *eclipse* in `org.eclipse.core.runttime.IPluginDescriptor` die beiden Methoden `getLabel()` und `getUniqueIdentifier` definiert.

- Plugins beider Systeme weisen Methoden zur Verwaltung des Plugin-Lebenszyklus auf, insbesondere zur Initialisierung: GstPlugin kennt eine `GstPluginInitFunc`, die beim Laden der entsprechenden Komponente aufgerufen wird. *eclipse* verwendet hierfür `startup()` aus `org.eclipse.core.runtime`.

Einen Unterschied stellt die Behandlung der Kommunikation zwischen Plugins dar: während in *eclipse* für jede Komponente mit der Spezifikation der *Extension Point*s im Manifest und der Dokumentation eine statische Beschreibung vorhanden ist, welche Arten der Erweiterung vorgesehen sind und wie diese auszusehen haben, geschieht der Datenaustausch bei GStreamer über typisierte Nachrichten. Dadurch wird bei GStreamer mehr Flexibilität zwischen den Komponenten erreicht, da sie nach Belieben hintereinander geschaltet werden können, wenn sie über die entsprechenden Pads verfügen. Dass dies möglich ist, liegt aber auch an dem Anwendungsgebiet des Multimedia-Streamings, da hier die Datenströme einen eingeschränkteren Typbereich abdecken, als dies bei *eclipse* vorgesehen ist und bei *Renew* nötig sein wird.

Aus dem gleichen Grund gibt es bei GStreamer auch eine klare Einteilung in Typen, in die die Elemente einzuordnen sind: Quellen, Senken und Filter. Da *eclipse*-Plugins in Funktionalität und Verhalten fast völlig frei sind, gibt es auch keine Möglichkeit, die Arten der Erweiterbarkeit zu definieren. Damit wird auch eine automatisierte vollständige Beschreibung der *Extension Point*s unmöglich. Als Resultat daraus ist deren Dokumentation für die Entwickler von Erweiterungen von großer Bedeutung.

4.2.4 Eignung für *Renew*

Nach der Vorstellung der beiden Pluginsysteme stellt sich die Frage, inwiefern eines davon für die Verwaltung von *Renew*-Komponenten geeignet ist.

GStreamer Wie schon in der Bewertung von GStreamer auf Seite 37 gesagt, ist GStreamer auf den Umgang mit Multimediadaten ausgerichtet. Dies resultiert in einer Struktur, die sich nicht für den Einsatz im Bereich von Softwarekomponenten eignet. Zusätzlich ist das GStreamer-System in der Sprache C geschrieben, während *Renew* in Java implementiert ist.

eclipse Bei *eclipse* handelt es durchaus um ein System, das für *Renew* interessant wäre. Von den ab Seite 42 dargelegten Nachteilen ist besonders das neue Window-Toolkit relevant, da ein großer Teil der *Renew*-GUI umgebaut werden müsste. Dies ließe sich vorläufig dadurch umgehen, dass die Bearbeitung der Netze weiterhin in eigenen Fenstern anstatt in die *eclipse*-IDE integriert geschieht.

Ein weiteres Problem besteht darin, dass *Renew* mit einer Eingliederung in *eclipse* stark daran angepasst werden müsste. Dies würde bedeuten, dass zukünftig erheblicher Aufwand betrieben werden müsste, um *Renew* neben der Verwendung als *eclipse*-Plugin als eigenständiges Programm lauffähig zu halten. Dafür reicht das für *Renew* verwendete Konfigurationsmanagement noch nicht aus. Dies spricht gegen die Einbindung von *Renew* als Plugins in *eclipse*.

Alternativ könnte der *eclipse*-Lader extrahiert und immer zum Start von *Renew* verwendet werden. Hierbei ergibt sich aber ein Lizenz-Problem: die *Common Public License*, unter der *eclipse* verteilt wird, erlaubt es nicht, Teile davon für ein Projekt zu verwenden, das – wie *Renew* – unter der *GNU Lesser General Public License LGPL* steht. Auch diese Möglichkeit steht also nicht zur Verfügung.

Als Ergebnis ergibt sich die Entscheidung, für *Renew* eine eigene Plugin-Verwaltung zu implementieren.

Kapitel 5

Visualisierung eines Plugin-Konzeptes

In dieser Arbeit wird Erweiterbarkeit als eine rekursive Eigenschaft eines Systems begriffen: das System als Ganzes wird von den in Kapitel 3 beschriebenen Komponenten erweitert, diese wiederum nach der Definition in Kapitel 4 durch Plugins.

Da Referenznetze Netze in Netzen erlauben, eignen sie sich gut als Modellierungstechnik für dieses Konzept: Eine Verwaltungsinstanz dient, ähnlich der Plattform in MULAN [Röl99], als Behälter für die Komponenten. Komponenten, die andere Komponenten als Erweiterung verwenden können, agieren diesen gegenüber ebenfalls als Plattform.

In diesem Kapitel wird ein Modell für erweiterbare Systeme vorgestellt, an dem sich die in den nachfolgenden Kapiteln beschriebene Umsetzung des Pluginsystem für *Renew* orientiert.

5.1 Allgemein

Um über Erweiterbarkeit reden zu können, ist es sinnvoll, eine Vorstellung davon zu haben, was damit gemeint ist. Sicherlich lassen sich verschiedene Arten der Erweiterbarkeit definieren; etwa die Erweiterbarkeit, die durch die Verwendung von Vererbung in objektorientierten Sprachen möglich ist. Dieses Kapitel stellt mit Hilfe von Referenznetzen ein Modell vor, das den Begriff der *Erweiterbarkeit* veranschaulicht.

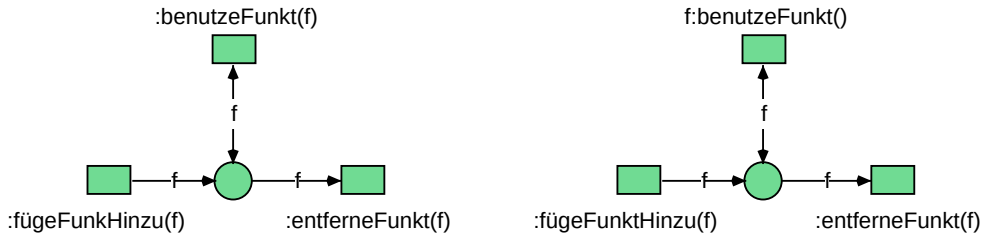
Abbildung 5.1 (a) stellt ein erweiterbares System auf allgemeine Art dar. Die Stelle in der Mitte ist dabei der Behälter, in dem alle Erweiterungen abgelegt werden. Diese können beliebig über die Transition auf der linken Seite hinzugefügt bzw. die Transition auf der rechten Seite entfernt werden. Außerdem kann vorhandene Funktionalität durch Schalten der oberen Transition ausgeführt werden.

Das Netz selbst wird hier als *Plattform* bezeichnet. Dieser Begriff geht konform mit dem der Plattform, wie er in MULAN verwendet wird.

Die Elemente, die auf die Stelle gelegt werden, sind die Komponenten in diesem System. Bei ihnen handelt es sich gemäß dem Netze-in-Netzen-Paradigma [VG03] ebenfalls um Referenznetze. Im folgenden Abschnitt wird das Thema behandelt, in wie fern die Komponenten in diesem Modell mit der Definition in Abschnitt 3.1 konform sind.

Bei dem Netz (a) ist zu bemerken, dass alle drei Transitionen darauf angewiesen sind, von außen aufgerufen zu werden. In der dargestellten Form hat es also am ehesten die Eigenschaften eines Behälters und bildet kein selbstständig ausführbares System.

Ein alternatives Netz zeigt Abbildung 5.1 (b). Der Unterschied an diesem Ansatz ist, dass die obere Transition einen Aufruf eines der enthaltenen Erweiterungen beinhaltet. Dadurch führt das System die vorhandenen Funktionen aus, wenn sich entsprechende Elemente auf der Stelle befinden.



(a) (b)

Abbildung 5.1: Modelle für erweiterbare Systeme

5.2 Komponenten

Das Modell auf Abb. 5.1 (b) stellt ein System dar, das die Funktionen von den enthaltenen Komponenten selbstständig verwenden kann.

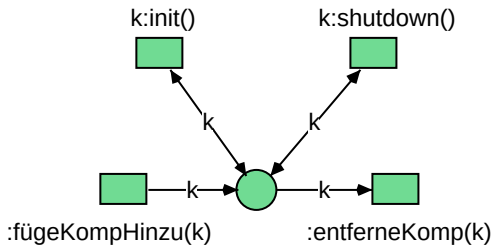
Das Problem dabei ist, dass die Schnittstelle der Elemente, die auf der Stelle liegen können, bereits durch das System vorgegeben sind: nur die Methoden, für die ein Aufruf im abgebildeten Netz vorgesehen ist, werden ausgeführt werden. Dadurch wird nur bestimmte, von Komponenten zur Verfügung gestellte, Funktionalität genutzt.

Mulan löst dies, in dem es die Schnittstelle der Agenten, die auf einer Plattform liegen können, möglichst klein hält: hier stehen lediglich `:send()` und `:receive()` zur Verfügung, mit denen Nachrichten verschickt werden können. Der Inhalt der Nachrichten bestimmt, welche Funktionen die Agenten ausführen sollen.

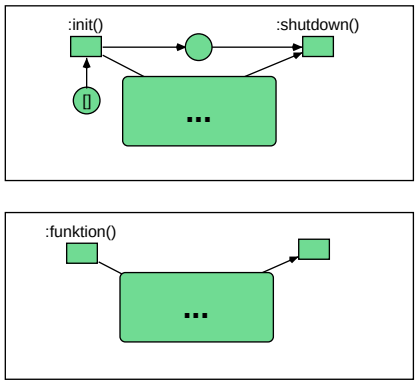
Hier soll ein ähnlicher Ansatz gewählt werden: Die ganze Funktionalität soll in der Komponente liegen und das System, wie es auf Abb. 5.1 gezeigt wird, lediglich der Verwaltung dienen. Als Möglichkeit werden *Start* und *Stop*-Methoden gewählt, die die Komponenten aktivieren bzw. deaktivieren. Damit ergibt sich als Plattformnetz das Netz auf Abbildung 5.2 (a). Die Methoden *Start*- und *Stop* heissen hier `init` respektive `shutdown`.

Auf der anderen Seite muss eine Komponente über die entsprechende Schnittstelle verfügen. Dies ist auf Abb. 5.2 (b) im oberen Rechteck dargestellt; der enthaltene Kasten mit den abgerundeten Ecken repräsentiert das Verhalten der Komponente. Der Zeitraum zwischen dem Aufruf der Methoden `init()` und `shutdown()` nennt sich *Lebenszyklus* der Komponente. Zu beachten ist ebenfalls, dass die `init`-Transition nur einmal aufgerufen werden darf. Dies wird durch das Token auf der eingehenden Stelle sicher gestellt.

Der untere Kasten auf Abbildung 5.2 (b) repräsentiert eine Schnittstelle der Komponente, die domänenspezifische Dienste zur Verfügung stellt, wie sie in der Definition von *Komponente* in Abschnitt 3.1 gefordert werden. Im gegenwärtigen Modell kann niemand diese Dienste nutzen: sie sind nur der Plattform bekannt, die lediglich die Lebenszyklus-Methoden aufruft. Damit eine Komponente den Dienst einer anderen verwenden kann, müssen sie miteinander kommunizieren. Diese Kommunikation wird in Abschnitt 5.4 erörtert.



(a) Modell für Plattform



(b) Lebenszyklus- und domänenspezifische Schnittstellen einer Komponente

Abbildung 5.2: Schnittstellen im Komponentensystem

5.3 Erweiterbarkeit von Komponenten

Im gegenwärtigen Zustand kann das Modell als erweiterbar auf einer Ebene bezeichnet werden. Dieser Begriff (*one-level extensible*, siehe [Szy02, Seite 95]) bezeichnet den Umstand, dass zwar neue Komponenten in das System eingebracht und ausgeführt werden können, sie selbst jedoch nicht erweitert werden können.

Genau dies zeichnet aber die Definition von *Plugin* in Kapitel 4 aus und soll daher ebenfalls hier konzeptionell besprochen werden. Um die Komponenten ebenfalls erweiterbar zu machen, müssen sie strukturell dafür vorgesehen sein. Dabei liegt es nahe, den gleichen Aufbau zu verwenden, der auch für die Plattform gilt, da diese ja bereits erweiterbar ist.

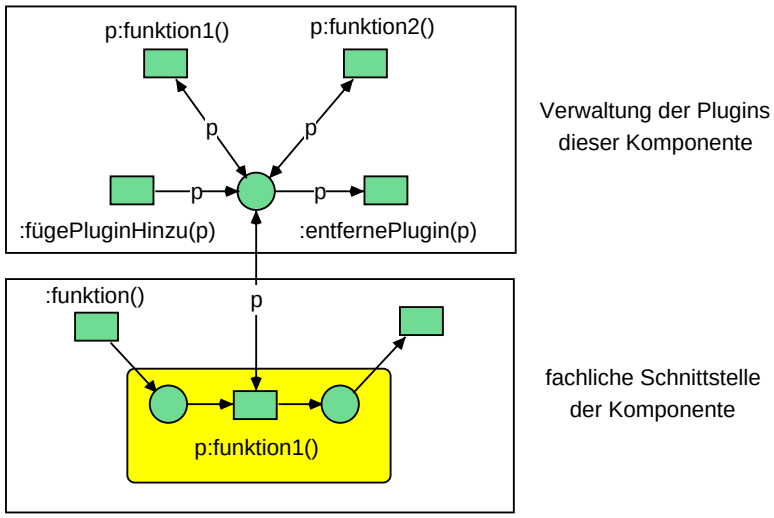


Abbildung 5.3: Modell für eine erweiterbare Komponente

Daraus resultiert für Komponenten ein Modell, wie es auf Abb. 5.3 dargestellt wird. Die Lebenszyklus-Schnittstelle ist hier weggelassen, um das Netz übersichtlicher zu halten.

Das obere Rechteck ist der Teil, in dem die Erweiterbarkeit analog zu dem Plattformnetz (vgl. Abb. 5.1 b)) umgesetzt wird. Die beiden Transitionen oben (`funktion1()` und `funktion2()`) sind die Aufrufe, die die Komponente an ihre Plugins richtet. Hierbei ist ersichtlich, dass die erweiterte Komponente nicht den Lebenszyklus des Plugins verwaltet, da sie nicht die auf der obersten Ebene definierten Methoden `init()` und `shutdown()` aufruft.

An dieser Stelle ist die Unterscheidung zwischen Komponente und Plugin vorzunehmen: ein Netz, das über die Lebenszyklus-Methoden verfügt, ist im Sinne dieses Modells eine Komponente. Ein Netz, das die Methoden `funktion1()` und `funktion2()` anbietet, ist zusätzlich in Bezug auf die Komponente in Abb. 5.3 ein Plugin.

Da jedes Plugin laut Definition in Abschnitt 4.1 ebenfalls eine Komponente ist, muss es ebenfalls die Lebenszyklus-Methoden zur Verfügung stellen, da es von der Plattform geladen und verwaltet wird. Durch diese Konstruktion kann das Plugin auch als gewöhnliche Komponente agieren und mehr als nur eine Komponente gleichzeitig erweitern.

In der Abbildung wird auch klar, dass jede Komponente nur durch bestimmte Arten

von anderen Komponenten sinnvoll erweitert werden kann, nämlich nur durch jene, deren Schnittstellen sie aufruft: im Beispiel `funktion1` und `funktion2`.

Die beiden Aufrufe, die die Komponente in Abb. 5.3 durch die beiden oberen Transitionen auf dem Plugin p ausführt, können zu einem beliebigen Zeitpunkt während des Lebenszyklus der Komponente, die p enthält, vorkommen, wodurch ihre Funktionalität erweitert wird. Dies ermöglicht es der Komponente, ebenfalls als Plattform zu agieren, wodurch die oben genannte Erweiterungsbeziehung von einer Ebene rekursiv auf beliebig viele Ebenen ausgedehnt werden kann, da jede Komponente wiederum andere Komponenten enthalten kann, die dann als Plugins bezeichnet werden.

Der untere Kasten von Abbildung 5.3 zeigt eine andere Form der Erweiterung: hier verändert das Plugin das Verhalten der erweiterten Komponente. Die Methode `funktion()` gehört zur Schnittstelle der Komponente, die in dem dargestellten Netz modelliert wird. Wird `funktion()` aufgerufen, löst dies ein Verhalten der Komponente aus, die in dem heller hinterlegten Kasten beispielhaft dargestellt ist. In diesem Beispiel greift die Funktion während ihrer Ausführung auf ein vorhandenes Plugin zurück. Je nachdem, welches dabei verwendet wird, ändert sich das Verhalten der abgebildeten Komponente.

Bekanntmachen der Erweiterung Wie werden nun Erweiterungen in die Komponenten eingefügt, für die sie bestimmt sind? Eine Möglichkeit besteht darin, dies die Plattform regeln zu lassen: So zeigt Abb. 5.4, wie eine Komponente hinzugefügt und den anderen Komponenten bekannt gemacht wird. Dabei muss die untere Transition `fuegePluginHinzu(p)` so oft schalten, wie Komponenten vorhanden sind; das dargestellte Netz ist an dieser Stelle aus Gründen der Übersicht vereinfacht.

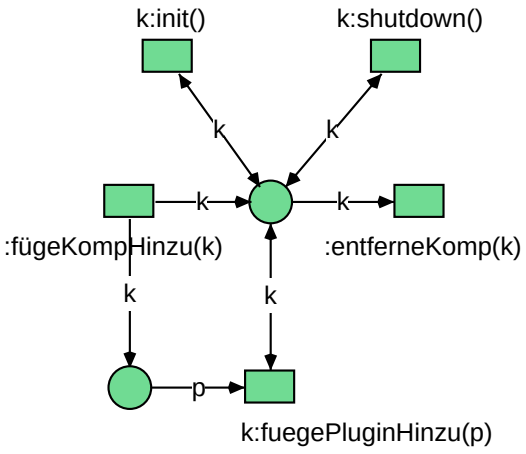


Abbildung 5.4: Die Plattform reicht eine neue Komponente als Plugin an die anderen

Alternativ dazu kann ein Plugin selbst dafür sorgen, die von ihm erweiterbaren Komponenten zu bestimmen. Diese Möglichkeit wird in dem später umgesetzten System verwendet. Dazu wird das im nächsten Abschnitt vorgestellte Konzept der *Dienste* verwendet.

5.4 Kommunikation zwischen Komponenten

Wie im Kapitel 3 beschrieben wird, ist ein Vorteil der Komponentenorientierung die Wiederverwendung. Das bedeutet, dass die von einer Komponente zur Verfügung gestellte Funktionalität von all denjenigen verwendet wird, die sie benötigen. Daher muss auch für die Komponenten eine Möglichkeit bestehen, auf andere Komponenten zuzugreifen. Wie kann dies in das Modell integriert werden?

Dazu bietet eine Komponente für jede Schnittstelle, die sie zur Verfügung stellt, eine Beschreibung dieser Schnittstelle an.

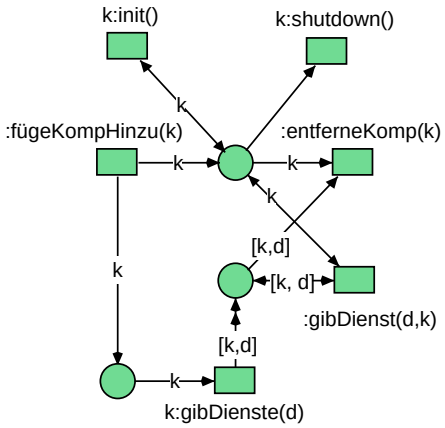


Abbildung 5.5: Modell für eine Plattform, die Dienste für Komponenten unterstützt

Auf Abbildung 5.5 ist eine Plattform zu sehen, die eine Verwaltung für Dienste beinhaltet. Hier wird beim Einfügen einer neuen Komponente eine Liste der Dienste gespeichert, die diese zur Verfügung stellt. Später kann die Komponente durch Angabe des gewünschten Dienstes abgerufen werden.

Der Pfeil mit der doppelten Spitze, der von dem Aufruf von `gibDienste()` ausgeht, bedeutet, dass mehr als ein Element auf die Stelle gelegt werde. Dadurch kann eine Komponente mehrere Dienste anbieten.

Zusätzlich ist zu beachten, dass Komponenten Zugriff auf die Plattform haben müssen, damit sie nach benötigten Diensten suchen können. Zu diesem Zweck muss die Komponentenschnittstelle um die Methode `gibDienste()` erweitert werden.

Erweiterung Im letzten Abschnitt wurde darauf hingewiesen, dass ein Plugin selbst dafür verantwortlich ist, die von ihm erweiterbaren Komponenten zu bestimmen. Dies ist eine weitere Funktion der Dienste: einen Dienst anzubieten bedeutet demnach nicht nur, eine bestimmte Schnittstelle zur Verfügung zu stellen, sondern auch einen bestimmten Typ von Plugins zu unterstützen. Dies entspricht dem Anbieten eines *Extension Point*s bei *eclipse*.

Die Erweiterbarkeit einer Komponente, die in Abb. 5.3 durch die Methode `fuegePluginHinzu()` umgesetzt wird, erfolgt in der Implementation durch eine gewöhnliche Schnittstelle; das heisst, ein Plugin fragt die Plattform nach den Anbietern eines bestimmten Dienstes und meldet sich dann über die dadurch erhaltenen Schnittstellen bei den Komponenten an.

5.5 Zusammenfassung

Dieses Kapitel stellt ein Modell für eine Plugin-Verwaltung vor, das die Grundlagen und Begriffe veranschaulicht, die in den vorherigen Kapiteln erarbeitet wurden.

Das Modell dient einerseits als Visualisierung, die die in einem Plugin-System vorkommenden Daten und Abläufe auf einfache Weise darstellt. Dies wird durch den Einsatz von Referenznetzen als Modellierungstechnik erreicht. Auf der anderen Seite bietet das Modell einen flexiblen Denkansatz, der das Konzept der Erweiterbarkeit beschreibt, wie es im folgenden Teil der Arbeit für *Renew* umgesetzt wird.

Hervorzuheben ist, dass dadurch, dass die vorgestellten Netze ebenfalls wieder Netze auf den Stellen enthalten können (*Netze-in-Netzen*), die Erweiterbarkeit eine rekursive Eigenschaft der gezeigten Elemente ist: Eine Komponente, die von einer Plattform geladen wird, kann einer anderen Komponente wieder als Plattform dienen. Dadurch entsteht eine Erweiterbarkeit auf mehreren Ebenen, was von SZYPERSKI genau als "Plugin-fähig" gesehen wird [Szy02, S.95].

Teil II

Ein Pluginsystem für *Renew*

Kapitel 6

Architektur von *Renew*

Um eine Aufteilung des bestehenden *Renew*-Systems in Komponenten vornehmen zu können, ist ein Überblick darüber notwendig, wie das Programm in seiner ursprünglichen Fassung aufgebaut ist. Dazu wird in diesem Kapitel eine Beschreibung der in *Renew* vorhandenen Java-*package*s gegeben, da diese eine gute Übersicht über das Gesamtprojekt bieten (vgl. Abschnitt 3.3).

Dieses Kapitel beschäftigt sich mit dem Grundsystem vor Beginn der Umstrukturierung. Dazu wird eine Liste derjenigen Java-Pakete sowie ihrer Funktionalität aufgestellt, deren Verständnis für den späteren Umbau des Systems von Interesse sind. Für eine genauere Beschreibungen siehe [KWD03]. Die hier verwendete Reihenfolge ist an den Paketabhängigkeiten orientiert; dabei werden die grundlegenderen zuerst behandelt.

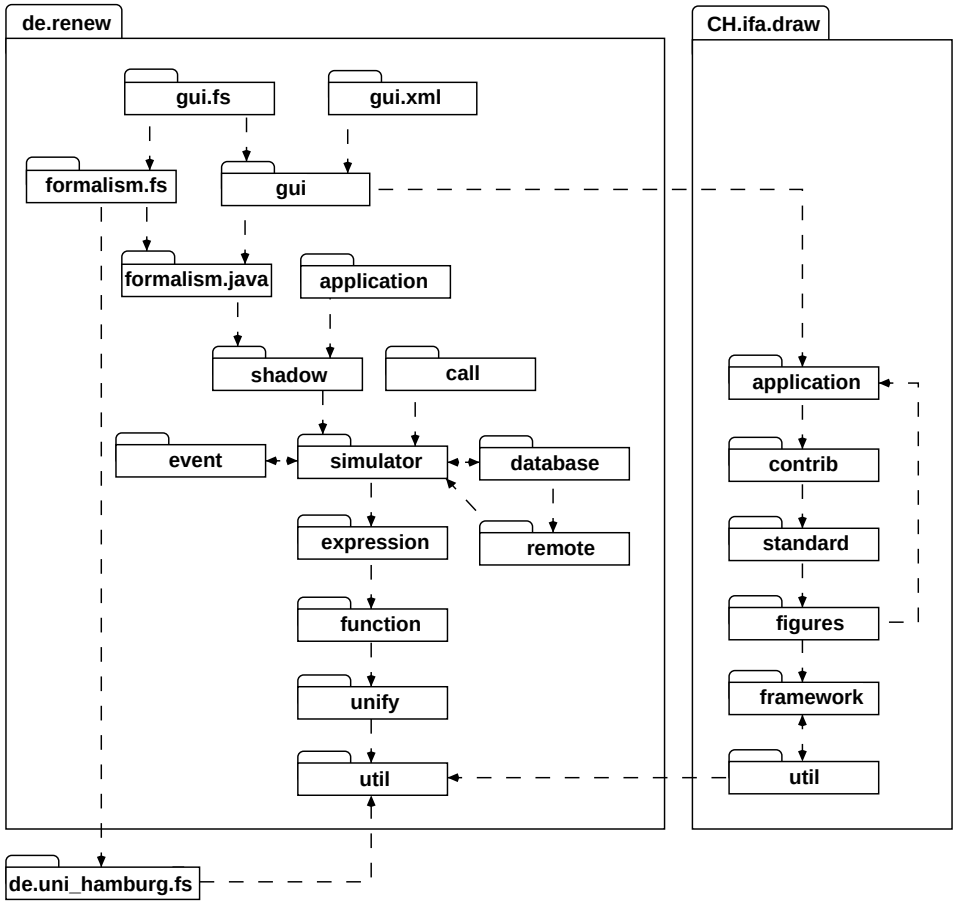


Abbildung 6.1: Die *package*s in *Renew* und ihre Abhängigkeiten

6.1 Übersicht

Abb. 6.1 zeigt die *package*s von *Renew* zusammen mit ihren Abhängigkeiten. Diese Grafik ist, ebenso wie die Beschreibung der *package*s, aus der *Renew*-Dokumentation [KWD03] entnommen. Die Darstellung zeigt den minimale Abhängigkeitsgraphen: da die Abhängigkeitsrelation transitiv ist, können viele der direkten Abhängigkeiten weggelassen werden. Dies entspricht mit Ausnahme der Abhängigkeit zwischen `simulator` und `database` der *transitiven Reduktion* [Tur96] des Graphen.

Wie in Abschnitt 8.1 zu sehen sein wird, fehlen hier sowohl einige Pakete als auch Abhängigkeiten zwischen Paketen. Dies ist jedoch für eine Erläuterung des Systems nicht maßgeblich, da es sich dabei um kleinere Abweichungen handelt.

6.2 Simulationskern

Der Simulationskern ist für die Simulation von Petrinetzen zuständig. Er enthält Datenstrukturen und Ausführungsalgorithmen, aber keine graphische Darstellung. Die folgenden Pakete gehören zum Kern:

de.renew.unify Dieses *package* enthält die Klassen des Unifikationsalgorithmus, der während der Simulation Variablenbindungen sucht.

de.renew.function Die Klassen in diesem Paket stellen ausführbare Funktionen im Sinne des Entwurfsmusters *Strategy* [GHJV95, Seite 315] dar. Dabei ist die Schnittstelle `Function` von zentraler Bedeutung, deren einzige Methode `function` zur Simulationszeit aufgerufen wird. Zusätzlich enthält das Package Implementationen für Java-Ausdrücke wie etwa `de.renew.function.CastFunction` oder `de.renew.function.InstanceofFunction`, mit denen die entsprechenden Operationen des *downcast*s resp. des `instanceof`-Operators implementiert werden.

de.renew.expression Dieses *package* fasst die Funktionalität der beiden vorigen zusammen, indem einerseits das Binden von Variablen als auch das Operieren auf Werten unterstützt werden. Außerdem führt es beispielsweise Konstanten sowie Typprüfung ein.

de.renew.simulator In den Klassen dieses *package*s befindet sich die Hauptfunktionalität des Simulators selbst. Er verwaltet die zu simulierenden Netze, alle Beschriftungen, Kanten und Transitionen.

de.renew.call Mit diesem *package* können Netze als Java-Objekte verwendet werden. Dies geschieht über generierte so genannte *Netz-Stubs*, die einen Aufruf an das entsprechende Netz weiterleiten.

de.renew.event Mit diesem Paket stehen Klassen für das Behandeln von Ereignissen zur Verfügung. Diese können beispielsweise durch das Schalten von Transitionen oder die Veränderung des Inhalts einer Stelle ausgelöst werden.

de.renew.database Dieses Package ermöglicht es, Zustand und Ablauf einer Simulation in einer Datenbank zu speichern. Damit kann eine Simulation persistent gemacht werden, so dass zu einem späteren Zeitpunkt ihr Zustand wieder hergestellt werden kann.

de.renew.remote In diesem Package befinden sich Klassen, die den Zugriff auf eine laufende Simulation über Java-RMI-Aufrufe erlauben.

de.renew.shadow Bei `shadow` handelt es sich um eine Zwischenschicht, die für die Trennung von Simulator und GUI sorgt: Shadow-Netze können ausgeführt werden, enthalten jedoch keine Information, wie ihre Elemente angezeigt werden sollen.

6.3 Formalismen

Die Formalismen-Pakete enthalten die Compiler, die dazu verwendet werden, die gezeichneten Netze in vom Simulator ausführbaren Code zu übersetzen. In Abschnitt 8.2 wird noch einmal genauer analysiert, welche Änderungen hier notwendig werden.

Dabei ist das maßgebliche Interface, das von den Compilern implementiert werden muss, `de.renew.shadow.ShadowCompiler`. Dieses deklariert `interface`s zum Übersetzen der Netze sowie für bestimmte Syntax-Überprüfungen.

de.renew.formalism.java Dieses *package* enthält den Standard-Compiler, der Referenznetze mit Java-Anschriften übersetzen kann.

de.renew.formalism.stub Bei diesem *package* handelt es sich um ein experimentelles Feature von *Renew*. Der Compiler, den es zur Verfügung stellt, ist ein Java-Net-Compiler, der um zusätzliche Typen für die Erzeugung von Netz-Stubs bereichert ist. Diese machen es möglich, Netze in einer Simulation als Java-Objekte zu verwenden.

de.renew.formalism.bool Dieser Formalismus ermöglicht Petrinetze, die als Anschriften Bool'sche Ausdrücke auswerten.

de.renew.formalism.fs und `de.renew.formalism.fsnet` unterstützen Feature Structures [Wie01].

6.4 Grafische Benutzungsoberfläche

Die grafische Benutzungsoberfläche ist ein wichtiger Bestandteil von *Renew*. Sie ermöglicht das Zeichnen von Netzen mit allen enthaltenen Elementen wie Transitionen, Stellen, Verbindungen sowie Anschriften.

Andererseits wird sie für die Simulation der Netze nicht benötigt. Die beiden großen Komplexe des Simulationskerns einerseits und der GUI andererseits wird bei der Zerlegung von *Renew* in Komponenten eine große Rolle spielen. Genaueres zu diesem Problem beschreibt Abschnitt 8.2.

Die für *Renew* verfügbaren Zeichenfunktionen basieren auf dem Open-Source Grafik-Framework *JHotDraw* (verfügbar unter [GE04]), das allerdings an vielen Stellen an spezielle Anforderungen von *Renew* angepasst wurde. Dabei wurde der ursprüngliche Java-*package* -Name `CH.ifa.draw` jedoch beibehalten.

de.renew.gui Dies ist der Kern der auf Basis von JHotDraw für *Renew* erstellten Grafikklassen. Sie nutzen dieses Framework, um Elemente und Werkzeuge zur Verfügung zu stellen, die für das Zeichnen von Petrinetzen sinnvoll sind. Außerdem sind in diese Klassen auch Funktionen zum Abspeichern der gezeichneten Elemente vorhanden. Informationen für die grafische Darstellung, wie etwa Bildschirmposition und Größe, werden in der oben genannten *Shadow*-Schicht nicht benötigt und stehen deshalb dort auch nicht zur Verfügung. In diesem *package* befindet sich auch die Klasse `de.renew.gui.CPNApplication`, die die `main`-Methode zum Starten von *Renew* enthält.

de.renew.gui.xml Klassen aus diesem *package* sind für die Speicherung von Netzen im xml-Format zuständig.

de.renew.gui.maria Dieses *package* unterstützt den Export gezeichneter Netze in das Petrinetz-Analysewerkzeug *Maria*.

de.renew.gui.fs Hier sind die Elemente und Werkzeuge zu finden, die für die oben genannten Feature Structures neu hinzugekommen sind.

Die in Abschnitt 3.3.2 bereits erwähnten *Mode* befinden sich ebenfalls in `de.renew.gui`. Dies liegt daran, dass sie Methoden für die Darstellung von Elementen während der Simulation bereit stellen.

Die andere Aufgabe der *Mode*s, die Bereitstellung des Compilers, gehört fachlich gesehen zu den Formalismen. Aus diesem Grund werden die *Mode*s im Zuge der Aufteilung von *Renew* in Komponenten in zwei Teile gespalten.

Kapitel 7

Entwicklung der Plugin-Verwaltung

Vor der Aufteilung von *Renew* in fachliche Komponenten ist es notwendig, das Verwaltungssystem zu entwerfen, das für das Laden und die Kommunikation zwischen den Komponenten verwendet wird. Dies geschieht in diesem Kapitel.

Dazu werden die Begriffe aus den Kapiteln 3 sowie 4 verwendet. Die Umsetzung orientiert sich an dem in Kapitel 5 vorgestellten Modell.

Am Ende dieses Kapitels steht als Ergebnis eine Plugin-Verwaltung, die es ermöglicht, generische Komponenten zu laden. Die Komponenten, die von dieser Plattform verwaltet werden, entstehen aus der Zerlegung des vorhandenen *Renew*-Systems. Der Vorgang dieser Zerlegung wird im nächsten Kapitel beschrieben.

7.1 Prototyp

Das Ziel der vorliegenden Arbeit ist der Entwurf eines Pluginsystems für *Renew*. Dabei geben die in den Kapiteln 3 und 4 beschriebenen Grundlagen eine gute Vorstellung, welcher Grundstock an Funktionalität dafür nötig sein wird.

Andererseits macht es wenig Sinn, zu viel Umbauarbeit auf einmal zu leisten: so besteht die Gefahr, dass Fehler oder Unzulänglichkeiten in das System geraten, deren Entfernung unnötig aufwändig ist.

Als Lösung dieses Problems bietet sich die Entwicklung einer vorläufigen Version des angestrebten Mechanismus an, der das grundliegende System minimal verändert. Insbesondere kann diese Arbeit schon geschehen, ohne die Aufteilung des Systems in Komponenten, wie sie in Kapitel 8 beschrieben wird, durchzuführen. Das Modell, wie es in Kapitel 5 dargestellt wird, wird in dieser Phase noch nicht umgesetzt.

Das Ziel ist es, technisch orientierte Erfahrungen zu gewinnen, die für das spätere System nützlich sein werden. Dieser Abschnitt beschreibt den bei *Renew* entwickelten Prototypen und führt anschließend auf, welche Erkenntnisse daraus für das angestrebte Gesamtsystem gewonnen werden können.

7.1.1 Anforderungen an den Prototypen

Als erstes soll die Frage geklärt werden, auf welche Vorraussetzungen hin die Entwicklung des Prototyps stattfindet. Dabei sind drei Punkte zu berücksichtigen.

Zuerst werden zwei Beispiel-Plugins vorgestellt. Diese dienen dazu, Anforderungen an den Prototyp fest zu legen und ihn zu evaluieren.

Zweitens ist eine grundlegende Systemvision wünschenswert, die eine Vorstellung davon liefert, was der Prototyp leisten können und auf welcher Ebene er sich in das System einfügen soll. Diese Frage ist konzeptioneller Natur und bietet den Rahmen für die weitere Entwicklung.

Drittens muss die in Kapitel 6 vorgestellte Architektur berücksichtigt werden. Dies beleuchtet die technische Seite der Vorbedingungen; als Ergebnis ist die Frage zu beanworten, an welcher Stelle das bestehende System modifiziert werden kann und muss.

Beispiel-Plugins

Zum Testen des zu entwickelnden Prototypen ist es sinnvoll, einige Plugins zur Verfügung zu haben, um die Erfahrungen bei deren Entwicklung evaluieren zu können.

An dieser Stelle spielt wieder die Ausrichtung von *Renew* auf MULAN, das Rahmenwerk zur agentenorientierten Softwareentwicklung, eine Rolle. Nach der Durchführung eines Lehreprojektes, in dem MULAN [Röl99] zum Einsatz kam, ergeben sich insbesondere zwei Stellen, an denen *Renew* in Hinsicht auf die Entwicklung mit MULAN verbessert werden kann.

NetComponents Der eine Bereich betrifft den Entwurf der Protokolle, die das Verhalten der Agenten regeln. Bei diesen handelt es sich um Referenznetze, wie sie von *Renew* zur Verfügung gestellt werden. Diese werden schnell sehr umfangreich und damit unübersichtlich; damit wird es schwer, sich in ein fremdes Protokollnetz einzuarbeiten, und auch eigene Netze sind ab einer gewissen Größe schwer zu überschauen.

Als Lösung dieses Problems wird in [Cab02] das Konzept der *NetComponents* entwickelt. Dabei handelt es sich um einen Satz vorgefertigter Netzelemente mit klarer Semantik, die sich bausteinartig zu einem Protokollnetz zusammenstellen lassen. Dies

schafft Einheitlichkeit in den Netzen und sorgt damit auch für eine eingängige grafische Repräsentation.

Da die Entwicklung der NetComponents parallel zu der Erstellung des Prototyps des Plugin-Mechanismus lief, lag es nahe, diese in *Renew* zu integrierenden Bausteine als Plugin zu entwerfen. Dies ist auch aus fachlicher Sicht sinnvoll, da nicht jeder Anwender die NetComponents benötigt, auch wenn ihre Nützlichkeit nicht auf die Verwendung innerhalb von MULAN beschränkt ist.

Die Anforderungen, die die NetComponents an das Pluginsystem stellen, sind folgende:

- Sie verfügen über ein Menü, das in das bestehende *Renew* eingegliedert werden soll.
- Sie müssen auf das aktuell dargestellte Netz schreibend zugreifen können, um eine bestimmte Netzkomponente dort einzufügen.
- Sie müssen eine Palette (auch *Toolbar* genannt) zu der Oberfläche von *Renew* hinzufügen können, damit der Benutzer die einzufügende Netzkomponente auswählen kann.
- Sie müssen auf das Dateisystem zugreifen können, um die zur Verfügung stehenden Netzkomponenten zu laden.

MulanViewer Die andere während des oben genannten Projektes ermittelte wünschenswerte Erweiterung von *Renew* ist ein Werkzeug, das es erlaubt, den Ablauf des mit MULAN entwickelten Systems zu beobachten [Car01]. Da *Renew* durch den Umfang an Netzen, der in MULAN enthalten ist, sehr viele Ausgaben produziert, ist eine sinnvolle Fehlersuche fast unmöglich.

Außerdem ist die Vielzahl an Ebenen, durch die MULAN gegliedert ist, besonders am Anfang verwirrend, so dass viel Zeit damit verloren geht, das Netz, das den Fehler verursacht, zu finden.

Beim *MulanViewer* handelt es sich um ein eigenständiges Fenster, der Dinge wie eine Liste der einzelnen im System vorhandenen Agenten, ihre aktivierten Protokolle und den Datenaustausch zwischen ihnen übersichtlich darstellt. Damit löst er die oben beschriebenen Probleme.

Für den MulanViewer gilt, ebenso wie für die NetComponents, dass er nicht von allen Benutzern benötigt wird. Seine Nützlichkeit beschränkt sich auf die Verwendung mit der MULAN-Entwicklung. Daher ist auch der MulanViewer ein guter Kandidat für ein Plugin.

Die Anforderungen, die er an den Plugin-Mechanismus stellt, sind folgende:

- der MulanViewer muss ein Menü in *Renew* einfügen können
- er muss Zugriff auf die Netze haben, die momentan ausgeführt werden. Aus ihrem Zustand liest er die Informationen, die für seine Anzeige wichtig sind.
- er muss ein eigenes Fenster öffnen können, in dem er die von ihm angebotene Sicht auf das Agentensystem anzeigt.

Systemvision

Wie soll der geplante Prototyp nun aussehen? Konzeptuell wird durch ihn lediglich eine Ebene der Erweiterbarkeit eingeführt: er ermöglicht das Laden von Komponenten, die die Schnittstellen der in *Renew* vorhandenen Objekte nutzen können. Eine Steuerung der Kommunikation wie in dem in Kapitel 5 vorgestellten Modell findet nicht statt.

Installation eines Plugins Zunächst ist es wichtig fest zu legen, in welcher Form der Benutzer neue Plugins in das System einbringen können soll. Hierbei ist das bei *eclipse* verwendete System eingängig: der Code eines Plugins befindet sich in einer jar-Datei, die zusammen mit einer Datei, die Metainformationen über das Plugins enthält, in ein Verzeichnis gelegt wird. Die Verzeichnisse mit den Plugins liegen dabei unterhalb des *Renew*-Installationsverzeichnisses.

Einbindung in die grafische Oberfläche Eine weitere Frage ist, wie sich zusätzliche Plugins in die vorhandene Benutzungsoberfläche einbetten. Beide vorgesehenen Erweiterungen, sowohl die NetComponents als auch der MulanViewer, benötigen einen Zugriff auf die grafische Benutzungsoberfläche von *Renew*: beide verfügen über ein Menü, um das das *Renew*-Menü ergänzt werden soll. Zusätzlich erzeugen die NetComponents eine Palette mit Schaltflächen, die ebenfalls angezeigt werden soll.

Der Prototyp soll das bestehende System möglichst wenig beeinflussen. Um den Unterschied zwischen Plugins und Basisfunktionalität auch optisch hervor zu heben, erscheinen die von der Plugin-Verwaltung erzeugten Menüeinträge in einem eigenen Menü.

Format der Metainformationen Eine weitere Frage ist das Format, in dem die Metainformation für ein Plugin abgelegt werden soll. Da es sich bei *eclipse* hierbei um Dateien in der *eXtensible Markup Language* XML handelt, kommt diese auch im Prototyp für *Renew* zur Anwendung. Zum Lesen dieses Formats gibt es bereits umfangreiche Java-Bibliotheken (siehe [Apa03] oder [IBM03a]), bei dem *Java Development Kit* ab der Version 1.4 wird eine Unterstützung bereits mitgeliefert.

Zudem ist XML gut für rekursive Strukturen geeignet. Dadurch ist es möglich, dass die Plugins den Aufbau ihres Menüs bereits in der XML-Datei festlegen. Dadurch kann das Pluginsystem selbst das Erstellen des Menü-Objektes übernehmen, die Datei parsen und ein entsprechendes Menü erzeugen und einfügen.

Technische Überlegungen

Neben den Überlegungen, wie der allgemeine Umgang mit dem vorläufigen Pluginsystem aussehen soll und welche Anforderungen es erfüllen soll, müssen auch technische Aspekte in Erwägung gezogen werden.

Laden der Plugins Dabei ist zunächst wichtig, sich vorstellen zu können, wie Komponenten in das System gelangen. Das bei Java enthaltene Framework unterstützt so genannte *Classloader*. Diese Objekte verwalten die Art und Weise, wie Klassen in das System geladen werden; für gewöhnlich geschieht dies aus dem Dateisystem. Wie oben erläutert, sollen auch die Plugins in lokalen Dateien liegen.

Wenn ein Java-Programm aufgerufen wird, wird der *classpath*, eine Umgebungsvariable, ausgewertet und die darin enthaltenen Verzeichnispfade zum Suchen nach zu ladenden Klassen verwendet. Der Unterschied zu den Plugins ist nun, dass sich deren

Verzeichnisse nicht im ursprünglichen *classpath* befinden, sondern vielmehr vom Pluginsystem lokalisiert werden.

Da die vorhandenen *Classloader* es nicht erlauben, zur Laufzeit einen neuen Pfad hinzuzufügen, muss zum Laden von Plugins eine eigener *Classloader* geschrieben werden. Dabei ist darauf hinzuweisen, dass die ursprünglichen *Renew*-Klassen, die nicht von dem Plugin-Loader geladen wurden, nicht auf dessen Klassen zugreifen können. Dies macht insbesondere nach der Aufteilung von *Renew* in Komponenten (Kapitel 8) die Einführung einer Abhängigkeitshierarchie nötig, die festlegt, in welcher Reihenfolge die Komponenten geladen werden müssen.

Architektur Außerdem stellt sich die Frage, wie der Code des Prototyps in das bestehende System eingebracht werden soll. Da sich das Laden eines Plugins nicht in die bisher in *Renew* vorhandenen Funktionalitäten eingliedern lässt, ist die Erstellung eines eigenen *package*s sinnvoll. Dies erleichtert ein späteres Entfernen des Prototypen. Um konform mit der bisherigen Namensgebung der *Renew-package*s zu bleiben, soll das *package* `de.renew.plugin` heißen.

Idealerweise sollte dieses Paket unabhängig von allen bisherigen sein. Aus den Anforderungen wird jedoch ersichtlich, dass die Plugin-Verwaltung auf die grafischen Klassen von *Renew* Einfluss nehmen können muss, um Menü- und Paletteneinträge vorzunehmen. Dadurch wird das neue *package* von `de.renew.gui` abhängig sein, da sich die entsprechenden Funktionen hier finden, genauer: in `de.renew.gui.CPNApplication`, da diese Klasse für die Erzeugung der Menüs und Paletten zuständig ist.

7.1.2 Aufbau des Prototyps

Nachdem die Rahmenbedingungen für den Prototypen eines *Renew*-Pluginsystem festgelegt sind, kann der konkrete Aufbau des Prototyps, der für diese Arbeit erstellt wurde, beschrieben werden.

Format der Metainformationen

Die Metainformationen eines Plugins befinden sich in dem XML-File `plugin.XML` im gleichen Verzeichnis wie der Code der entsprechenden Komponente. Diese Datei hat zwei hauptsächliche Funktionen: einerseits das Plugin zu beschreiben, andererseits, die Menüstruktur für die GUI zur Verfügung zu stellen. Als Beispiel soll eine leicht abgewandelte Datei der `NetComponents` dienen. Abbildung 7.1 zeigt *Renew* mit dem Menü, das aus dem Beispiel entsteht.

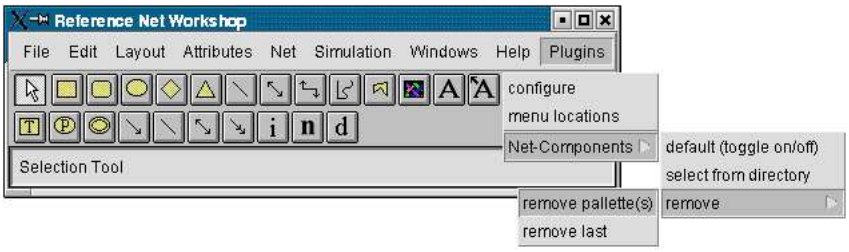


Abbildung 7.1: *Renew* mit Plugin-Menü des Prototyps

```
<?XML version="1.0" encoding="UTF-8"?>

<plugin
(1)     name="Net-Components"
        className="de.renew.netcomponents.ComponentsToolPlugin">

<!-- The description of the plugin's menu.
     It will appear in the "plugins"-menu under
     the plugin's name.-->
<menu>
(2)
  <!-- creates a menuitem with the label "item"
       which will result in calling the given class'
       execute()-method when clicked
       (class must be an instance of
        de.renew.PluginCommand).
    -->
  <menuitem
(3)     label="default (toggle on/off)"
(4)     command=
         "de.renew.netcomponents.
          ShowNetComponentsToolCommand"/>
  <menuitem
        label="select from directory"
        command=
         "de.renew.netcomponents.
          SelectNetComponentsToolCommand"/>
  <menu label="remove">
    <menuitem
          label="remove pallette(s)"
          command=
           "de.renew.netcomponents.
            RemoveNetComponentsToolCommand"/>
    <menuitem
          label="remove last"
          command=
           "de.renew.netcomponents.
            RemoveLastNetComponentsToolCommand"/>
  </menu>
</menu>

</plugin>
```

An der Stelle (1) befindet sich die Information über das Plugin in Form von Attributen des Tags `plugin`. Dabei gibt `name` einen eindeutigen, von Menschen lesbaren Bezeichner an. Unter `className` steht der Name der Klasse, die das zu dieser Datei zugehörige Plugin vertritt. Diese muss das Interface `de.renew.plugin.IPlugin` implementieren und einen Konstruktor ohne Argumente haben, um vom System instanziiert werden zu können.

Bei (2) beginnt der Menü-Teil der XML-Datei. Ein `<menu>`-Eintrag kann zwei Arten von Untereinträgen haben: entweder wieder `<menu>` oder `<menuitem>`. Beide haben als Attribut einen `Label` (3); dies ist der String, der im Menü angezeigt wird. Ein `menuitem` hat zusätzlich noch ein `command`-Attribut (4). Hier muss eine Unterklasse von `de.renew.plugin.PluginCommand` angegeben werden, dessen Methode `execute` ausgeführt wird, wenn der Benutzer den entsprechenden Menübefehl auswählt.

Klassenmodell

Abbildung 7.2 zeigt ein Klassendiagramm des Prototypen. Dabei sind die beiden Klassen auf der linken Seite, `PluginManager` und `PluginLoader`, diejenigen, die das Pluginsystem ausmachen, da sie für das Laden und Verwalten der Plugins verantwortlich sind.

Auf der rechten Seite sind die Interfaces `IPlugin`, `IMenuOwner` sowie `PluginCommand`. Bei diesen handelt es sich um die Schnittstellen, die benötigt werden, um die Funktionalität der Plugins verfügbar zu machen.

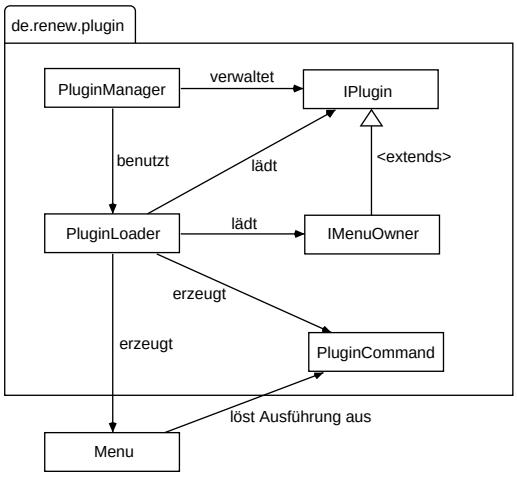


Abbildung 7.2: Klassendiagramm des Prototyps

Ablauf des Ladens

Die Abfolge der Methodenaufrufe zwischen den Exemplaren der Klassen ist in Abbildung 7.3 gezeigt. Darauf ist zu sehen, dass der `PluginManager` das Laden der Komponenten an den `PluginLoader` delegiert. Dieser parst dann die gefundenen **plugin.XML**-Dateien und erstellt aus den darin enthaltenen Informationen die Plugins sowie deren Menüs.

Konfiguration der Plugins

Der letzte Punkt, der für den Prototypen des Pluginsystems angesprochen werden soll, ist der der Konfigurierbarkeit. Wie auf dem Screenshot in Abb. 7.1 zu sehen ist, gibt es in dem Plugin-Menü einen Punkt `configure`.

Wird dieser geklickt, öffnet sich ein Dialogfenster, in dem eine Liste der geladenen Plugins dargestellt ist. Aus dieser kann sich der Benutzer auswählen, welches Plugin er konfigurieren möchte. Das System ruft dann die Methode `getConfigurationComponent`

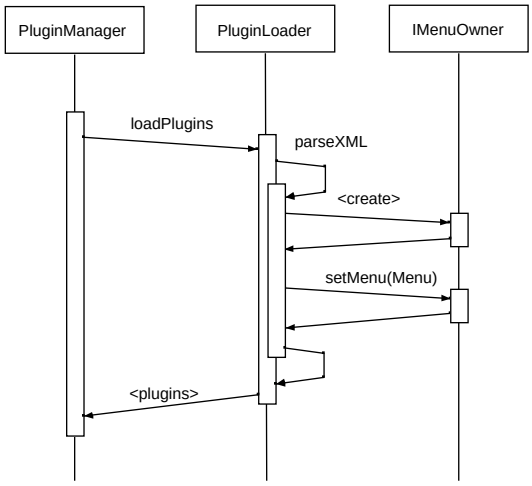


Abbildung 7.3: Sequenzdiagramm des Ladevorgangs beim Prototyp

aus dem Interface `IPlugin` auf. Diese liefert eine Oberflächenkomponente vom Typ `java.awt.Component`, die im Dialog dargestellt wird. Es liegt beim Plugin, mit der Komponente die Konfigurationsdaten auszutauschen. Diese Konfigurationsmöglichkeit wurde von keinem der Plugins genutzt.

7.1.3 Ergebnis

Der vorgestellte Prototyp wurde erstellt, um aus seiner Benutzung praktische Erfahrungen zu sammeln, die in die Entwicklung des endgültigen Systems einfließen. Daher folgt jetzt eine Diskussion der Punkte, die den erklärten Mechanismus ausmachen, und ob und wie sie sich in der Praxis bewährt haben. Dies geschieht sowohl unter technischen Aspekten als auch hinsichtlich Benutzbarkeit seitens der Plugin-Entwickler.

Erfüllung der Anforderungen

In der Einleitung werden an das zu entwickelnde System allgemeine Anforderungen gestellt. Diese bestehen darin, dass *Renew* im Hinblick auf Erweiterbarkeit, Konfigurierbarkeit und Verwaltbarkeit verbessert werden sollte. Es liegt daher nahe, die Eigenschaften des Prototypen auf diese Aspekte hin zu untersuchen.

Erweiterbarkeit Als erstes soll die Frage gestellt werden, ob der vorgestellte Prototyp es ermöglicht, *Renew* um Funktionalität zu erweitern. Dies ist sicherlich der Fall. Beide für den Protoypen entwickelten Plugins, sowohl die NetComponents als auch der MulanViewer, bereichern das System um vorher nicht vorhandene Funktionen.

Diese können mit dem Pluginsystem eingefügt werden, ohne dass eine Veränderung des bestehenden Programmcodes notwendig gewesen sei. Diese Anforderung kann also als erfüllt bezeichnet werden.

Konfigurierbarkeit Ist *Renew* durch die Einführung des prototypischen Pluginsystems konfigurierbarer geworden; ermöglicht es also dem Benutzer, die Arbeitsweise des Systems seinen Bedürfnissen anzupassen?

Diese Frage ist schwieriger zu beantworten als die nach der Erweiterbarkeit. Zunächst ist es möglich, Plugins, die nicht benötigt werden, nicht mit ins System zu laden. Dies kann als eine Form der Konfigurierbarkeit betrachtet werden.

Zum anderen können, wie oben erwähnt, Konfigurationseinstellungen für die einzelnen Plugins vorgenommen werden. Dies betrifft jedoch wirklich nur die Plugins; das Verhaltens von *Renew* selbst, etwa des Netzcompilers oder des Simulators, kann hierdurch nicht verändert werden. Insofern kann nicht gesagt werden, dass der Prototyp eine zusätzliche Konfigurierbarkeit in das System bringt.

Verwaltbarkeit Als letzter Punkt steht zur Diskussion, ob die Einführung des vorher beschriebenen Pluginsystems die Verwaltung von *Renew* als Softwareprojekt erleichtert.

Dazu ist zu sagen, dass neu erstellte Plugins einfacher zu verwalten sind, das zu Grunde liegende System jedoch unverändert bleibt. Dieses hat jedoch bereits einen Komplexitätsgrad erreicht, der eine Vereinfachung des Kerns wünschenswert macht. Zum anderen ist auch das Kernsystem Veränderungen unterworfen. Nicht alle Entwicklungsarbeit wird sich also in Plugins auslagern lassen.

Die Verwendung des prototypischen Pluginsystems unterstützt das Konfigurationsmanagement von *Renew* also nur minimal. Zudem lässt sich feststellen, dass die Installation der Plugins durch Kopieren der entsprechenden Dateien an die richtigen Stellen, umständlich zu benutzen ist. Dies kann behoben werden, indem eine Möglichkeit zu Automatisierung dieser Schritte zusätzlich als Anforderung an das Zielsystem aufgenommen werden.

Bewertung des Prototypen

Der auf Seite 62 genannte Zweck des Prototyps ist das Sammeln von Erfahrungen. Welche der angedachten Konzepte und Features werden nun benötigt und welche nicht? Welche zusätzlichen Anforderungen lassen sich aus der Entwicklung der Plugins für den Prototypen ableiten?

Um diese Fragen zu beantworten, sollen hier die drei wichtigsten Punkte beschrieben werden, die beim Umgang mit dem einfachen Pluginsystem auffallen. Diese beziehen sich spezifisch auf die Entwicklung der Plugins und nicht auf die allgemeinen Anforderungen, die oben behandelt werden.

Integration der GUI-Elemente im Pluginsystem Die Designentscheidung, die Struktur der Menüs in die Konfigurationsdatei der Plugins zu legen, ist problematisch. Da die Plugin-Verwaltung die Menüs erstellt, muss sie mit der grafischen Benutzungsoberfläche zu arbeiten. Dies ist bei der bisherigen monolitischen Struktur unproblematisch. Bei der vorgesehenen Komponentisierung (siehe Kapitel 8) ist die Plugin-Verwaltung für das Laden aller Komponenten zuständig und darf daher von keinem der Pakte abhängig sein.

Als Lösung muss die GUI Erweiterungsschnittstellen erhalten, die es anderen Komponenten erlauben, Menüs hinzuzufügen. Das dafür notwendige Vorgehen beschreibt Kapitel 9.2.

Konfigurationsdialog Wie beschrieben gibt es in der Schnittstelle, die ein Plugin implementieren muss, eine Methode, die eine grafische Komponente für die Konfiguration dieses Plugins liefert.

Hier liegt das gleiche Problem vor wie bei der Erstellung der Menüs: auch dieses Feature macht das Pluginsystem von der Benutzung einer grafischen Oberfläche abhängig. Zudem verwenden weder das NetComponents-Beispiel- Plugin noch der MulanViewer diese Möglichkeit. Aus diesen beiden Gründen sollten andere Methoden zur Konfiguration der Komponenten in Betracht gezogen werden.

Umgang mit der XML-Konfigurationsdatei Die Idee, für die Konfiguration der Plugins XML als Dateiformat zu verwenden, basiert auf zwei Gründen: zum einen unterstützt Java das Lesen dieses Formats, zum anderen ist die XML gut geeignet, rekursive Strukturen darzustellen, was für die Speicherung von Menüs sinnvoll ist.

Da sich die Unterstützung von Menüs auf Ebene des Pluginsystem als ungünstig erweist, fällt dieser Grund für die Verwendung von XML weg. Zudem zeigt sich für die Erstellung von XML-Dateien, dass XML schon bei den einfachen Beispielkonfigurationen der NetComponents und des MulanViewers für die Entwickler unhandlich zu verwenden ist.

Dies könnte durch den Einsatz von Werkzeugen zur Erstellung dieser Dateien erleichtert werden. Alternativ sollte ein anderes Format gesucht werden, das den einfacheren Anforderungen der Konfiguration ohne Darstellung von Menüs genügt.

Zusammenfassend kann gesagt werden, dass die mit dem Prototyp gemachten Erfahrungen wertvoll für die Entwicklung des endgültigen Plugin-Verwaltungssystems sind und den verhältnismäßig geringen zusätzlichen Entwicklungsaufwand rechtfertigen.

7.2 Spezifikation und Entwurf

Dieser Abschnitt beschreibt die Umsetzung des in Kapitel 5 entwickelten Konzeptes für *Renew*. Dazu wird spezifiziert, wie die einzelnen dort vorgestellten Aspekte umgesetzt werden können. Im Gegensatz zum Prototyp kann die hier vorgestellte Plugin-Verwaltung mit generischen Komponenten umgehen und ist nicht auf *Renew*-spezifische beschränkt.

7.2.1 Dienste

Dienste stellen die Funktionalität einer Komponente für andere zur Verfügung. Dazu meldet eine Komponente die von ihr bereitgestellten Dienste mit einer Dienstbeschreibung bei der Verwaltungsinstanz an.

Es muss also einerseits festgelegt werden, wie eine Dienstbeschreibung auszusehen hat, und andererseits, wie eine Komponente auf einen Dienst zugreifen kann, den sie bei der Verwaltung abgefragt hat.

Dienstbeschreibung

In der *eclipse*-Plattform wird die Dienstbeschreibung über das Manifest angeboten: der Eintrag `id` entspricht dem Namen eines Dienstes, auf den sich andere Plugins mit einem `requires`-Tag beziehen können (vgl. Abschnitt 4.2.2).

Die Dienstbeschreibung besteht also aus einem einzelnen String. Diese Umsetzung reicht aus und ist einfach zu bedienen. Daher soll sie auch für die *Renew*-Plugins eingesetzt werden. Zu beachten ist jedoch, dass Dienstnamen eindeutig sein müssen; dafür sorgt die Verwendung von Präfixen, die sich an der URL des Komponentenentwicklers orientieren, wie es schon bei Java-Packagenamen üblich ist.

Schnittstellenobjekte

Was es bedeutet, einen Dienst anzubieten, ist in *eclipse* der Dokumentation des entsprechenden Plugins zu entnehmen. Für gewöhnlich steht dadurch ein definierter Satz von Java-Klassen zur Verfügung. Plugin und Dienst stehen in *eclipse* in einer 1:1-Beziehung: ein Plugin stellt genau einen Dienst zur Verfügung, und ein Dienst kann nur von diesem Plugin angeboten werden.

Szyperski [Szy02] nennt zwei Möglichkeiten, die Schnittstelle einer Komponente zu beschreiben: direkte (prozedurale) oder indirekte (Objekt-) Schnittstellen. Zur zweiten Möglichkeit sagt er:

> Most component implementations can be made to have their interfaces look like object interfaces. This is the traditional "workaround" of pure object-oriented languages, like Java. [Szy02, Abschnitt 5.1.1]

Auch in dem hier angestrebten System wird diese "traditionelle" Möglichkeit gewählt: die Schnittstellen der Komponenten bestehen aus Objekten, die einen Dienst bereitstellen. Erfragt eine Komponente einen Dienst von der Verwaltung, so erhält sie ein Exemplar einer bestimmten Klasse; um welche es sich handelt, bestimmt der Dienstanbieter.

Um dieses Konzept umzusetzen, ist das Entwurfsmuster *Facade* (deutsch *Fassade*) [GHJV95, Seite 185] geeignet. Abbildung 7.4 (ebenfalls [GHJV95]) zeigt, wie ein Subsystem durch eine neu entstandene Klasse, die Fassade, gekapselt wird. Diese Struktur sei geeignet,

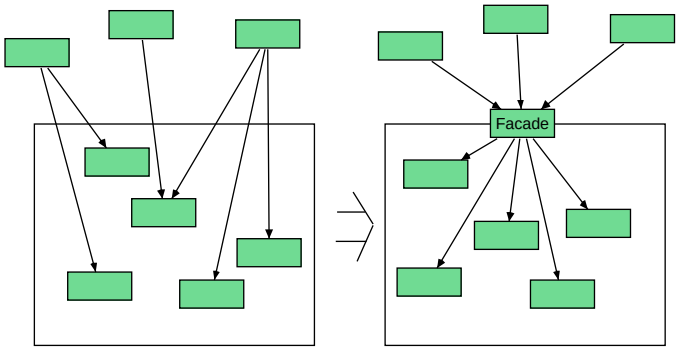


Abbildung 7.4: Struktur des Facade-Entwurfsmusters

> eine einheitliche Schnittstelle für eine Menge von Schnittstellen innerhalb eines Subsystems zur Verfügung zu stellen. Fassade definiert eine Schnittstelle auf höherer Ebene, wodurch das Subsystem einfacher zu benutzen ist. [GHJV95, S. 185]

Im vorliegenden Fall ist der Zweck nicht, die Benutzung der Schnittstelle zu vereinfachen, sondern die Kommunikation zwischen den Komponenten zu regeln. Ein Dienstname wird auf eine Java-Klasse oder ein Java-Interface abgebildet. Die Verwaltung in Abb. 7.5 liefert also ein Fassade-Objekt des Typs zurück, der dem angeforderten Dienst entspricht. Das Objekt heisst *Repräsentant* oder *Schnittstellenobjekt* der Komponente.

Eine Komponente kann mehr als eine Schnittstelle anbieten. In diesem Fall sind alle über das gleiche Repräsentanten-Objekt zugreifbar. Um den Repräsentant der Komponente ebenfalls für die Steuerung ihres Lebenszyklus verwenden zu können (s.u.), muss er mindestens das Interface `de.renew.plugin.IPlugin` implementieren.

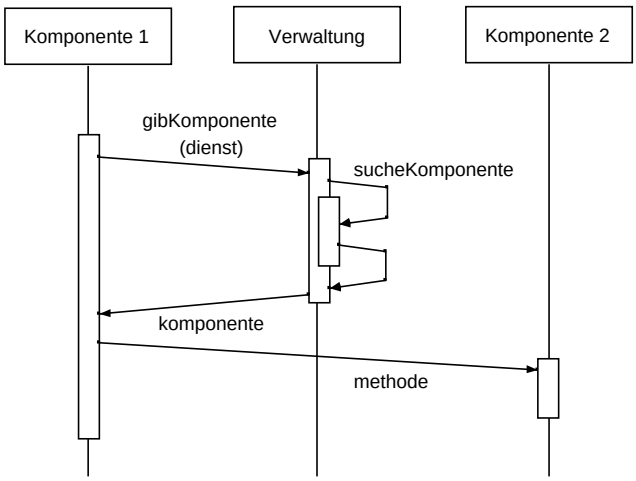


Abbildung 7.5: Abfrage einer Komponente nach Dienst

7.2.2 Abhängigkeiten

In Abschnitt 3.1 wird der Begriff der *Abhängigkeit* eingeführt. Dieser besagt, dass eine Komponente zur Erfüllung ihrer Dienste voraussetzt, dass bestimmte andere Dienste zur Verfügung stehen.

Diese Information wird vom Pluginsystem benötigt, um zu entscheiden, ob eine Komponente geladen werden kann: wenn eine Abhängigkeit nicht erfüllt ist, ist die Komponente nicht funktionsfähig und darf nicht ins System eingefügt werden.

Aus diesem Grund ist es auch hier sinnvoll, die Abhängigkeiten einer Komponente in dessen Metainformationen aufzunehmen. Dies stimmt auch mit dem in *eclipse* verwendeten Ansatz überein: hier gibt es ein `required`-Tag im Manifest der einzelnen Plugins. Ebenso wie eine Komponente mehrere Dienste anbieten kann, kann sie auch mehrere in Anspruch nehmen.

7.2.3 Lebenszyklus einer Komponente

Der Lebenszyklus einer Komponente beginnt, wenn sie in das System geladen wird und endet, wenn sie entladen wird. Das Zustandsdiagramm auf Abb. 7.6 zeigt das Modell des Lebenszyklus.

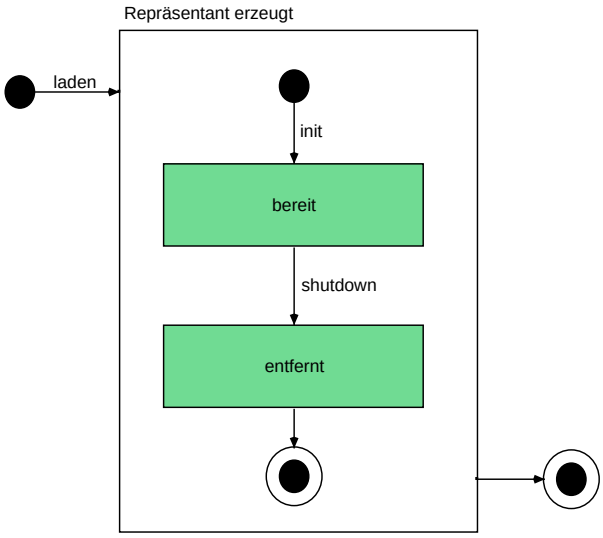


Abbildung 7.6: Lebenszyklus einer Komponente

Wie oben erwähnt, erfolgt die Steuerung des Lebenszyklus einer Komponente über ihr Repräsentantenobjekt. Die Plugin-Verwaltung erzeugt dieses anhand der in den Metainformationen gegebenen Daten. Der eigentliche Lebenszyklus beginnt mit dem Aufruf der Methode `init()`. Zu diesem Zeitpunkt ist sichergestellt, dass die von der Komponente deklarierten Abhängigkeiten erfüllt sind.

Der Aufruf der `shutdown`-Methode beendet den Lebenszyklus. Auch hier stellt die Plugin-Verwaltung sicher, dass keine andere Komponente von den Diensten der zu entladenden abhängig ist. Für die Komponente bedeutet dies, dass sie eventuell reservierte Ressourcen wieder freigeben, andere benötigte Aufräumarbeiten leisten und Erweiterungen, die sie in andere Komponenten eingebracht hat, wieder entfernen muss.

7.2.4 Laden eines Plugins

Damit ein Plugin in das System integriert werden kann, muss es zunächst geladen werden. Es folgt eine Beschreibung der Schritte, die in dem für *Renew* entwickelten System ausgeführt werden. Dabei wird der Punkt `laden` auf Abb. 7.6 feiner aufgegliedert. Am Ende dieses Schrittes steht das Repräsentantenobjekt, mit dem Lebenszyklus der Komponente verwaltet wird. Abbildung 7.7 zeigt ein Modell des Ladevorgangs mit den dabei entstehenden Daten.

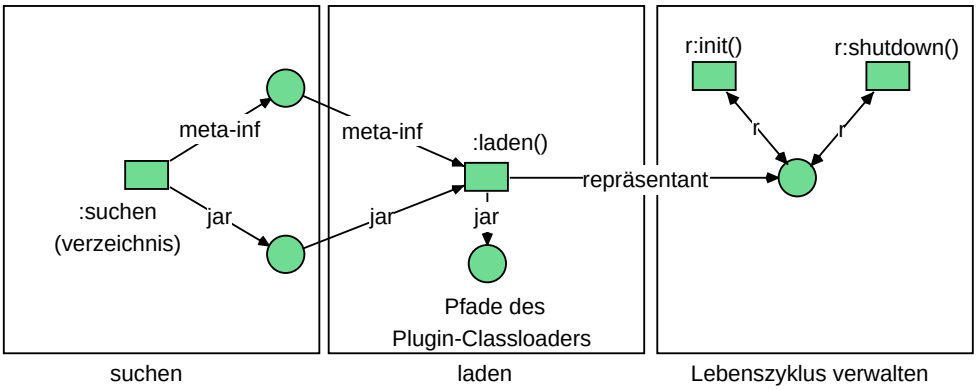


Abbildung 7.7: Laden einer Komponente: Ablauf

Suchen

Das in Abschnitt 7.1.2 für den Prototypen beschriebene Konzept, wie eine Komponente zur Verfügung gestellt wird, wird beibehalten: eine Datei mit Metainformationen und eine jar-Datei mit dem Programmcode liegen in einem gemeinsamen Verzeichnis.

Um nun ein Exemplar der Schnittstellenklasse erzeugen zu können, müssen erstens die durch die Komponente definierten Klassen zur Verfügung stehen. Zweitens müssen die Komponenten vorhanden sein, die von der zu ladenden Komponente als *required* deklariert sind (von denen sie also abhängig ist).

Um die zweite Bedingung zu prüfen, muss das Pluginsystem über die Metainformationen der Komponente verfügen. Wenn die Abhängigkeiten der Komponente nicht erfüllt sind, muss der Ladevorgang abgebrochen werden. Daher ist es sinnvoll, die Datei mit den Metainformationen zuerst zu lesen.

Als nächstes muss der zugehörige Programmcode bereit gestellt werden. Dies geschieht, wie in Abschnitt 7.1.1 beschrieben, über ein Hinzufügen der jar-Datei in den `Classloader` des Pluginsystems.

Laden der Klassen

Ebenfalls aus den Metainformationen ergibt sich mit dem **mainClass**-Eintrag die Klasse, die das System als Repräsentanten der Komponente instanziieren muss.

Dies kann nun geschehen, da die Abhängigkeiten überprüft sind und die erforderlichen Klassen der Komponente zur Verfügung steht. Konzeptionell bedeutet dies den Zustandsübergang der Komponente, der auf Abb. 7.6 mit *laden* gekennzeichnet ist.

Verwalten des Lebenszyklus

Da jetzt das Schnittstellenobjekt des Plugins zur Verfügung steht, kann die Verwaltung des Lebenszyklus durch das Pluginsystem beginnen, das heisst, dass es die dafür vorgesehene Methode `init()` aufruft.

Anschließend ist das Pluginsystem dafür zuständig, durch Aufruf der `shutdown()`-Methode den Lebenszyklus einer Komponente zu beenden, wenn sie entladen wird.

7.2.5 Laden mehrerer Plugins bei Systemstart

Der oben beschriebene Ablauf gilt für das Laden einer einzelnen Komponente; sind ihre Abhängigkeiten dabei nicht erfüllt, wird sie nicht geladen. Dies ist beim Start des Systems ungünstig, da hier mehrere Plugins auf einmal geladen werden müssen. Auch hier ist es wichtig, die Abhängigkeiten einzuhalten. Da aber die Reihenfolge, in der Komponenten gefunden werden, nicht festgelegt ist, müssen diese vor dem Laden sortiert werden. Dazu reicht es aus, deren Metainformationen zu berücksichtigen, da in ihnen die Abhängigkeitsbeziehungen enthalten sind.

Daraus ergibt sich für den Systemstart ein dreistufiger Prozess:

1. Lokalisieren der Komponenten und Erzeugen der Metainformationen
2. Sortieren der Metainformationen nach Abhängigkeiten
3. Laden der Komponenten in der entstandenen Reihenfolge

7.2.6 Daten und Transformationen

In der vorangehenden Beschreibung treten folgende Daten im Pluginsystem auf, die während des Systemablaufs ineinander überführt werden müssen:

- Ein **Ort**, an dem die Komponenten gesucht werden sollen. Dabei kann es sich um ein Verzeichnis handeln oder im allgemeinen um eine `URL`.
- Die **Metainformationen**, die zu einer Komponente gehören.
- Der **Programmcode**, die zu einer Komponente gehören.
- Die Objekte, die die Komponenten repräsentieren.

Folgende Transformationsschritte führt das System zwischen diesen aus:

1. Aus dem Ort wird eine Liste generiert, die die Metainformation der gefundenen Komponenten enthält
2. Diese Liste wird anhand der darin enthaltenen Information über die Abhängigkeiten sortiert
3. Die sortierte Liste wird iteriert und für jedes Metainformations-Objekt der zugehörige Programmcode geladen und ein Repräsentantenobjekt erzeugt
4. Die Plugin-Verwaltung beginnt den Lebenszyklus der Komponente

7.2.7 Aufteilung der Plugin-Verwaltung in Klassen

Für die verschiedenen Funktionen, die die Plugin-Verwaltung zu erfüllen hat, soll für jeden der Transformationsschritte im vorigen Abschnitt eine Klasse verantwortlich sein. Dabei muss eine hauptverantwortliche Klasse vorhanden sein, die darauf achtet, dass die Schritte in der richtigen Reihenfolge ausgeführt werden.

Diese Klassen sind auf dem Diagramm in Abb. 7.8 dargestellt. Dabei entspricht der `PluginManager` der Plattform, der die Komponenten beinhaltet. Er ist also für die Verwaltung der Lebenszyklen der Plugins sowie für ihre Kommunikation untereinander verantwortlich.

Die anderen Klassen `Finder`, `Loader` und `DependencyList` sind dem Manager untergeordnet und werden von diesem gesteuert.

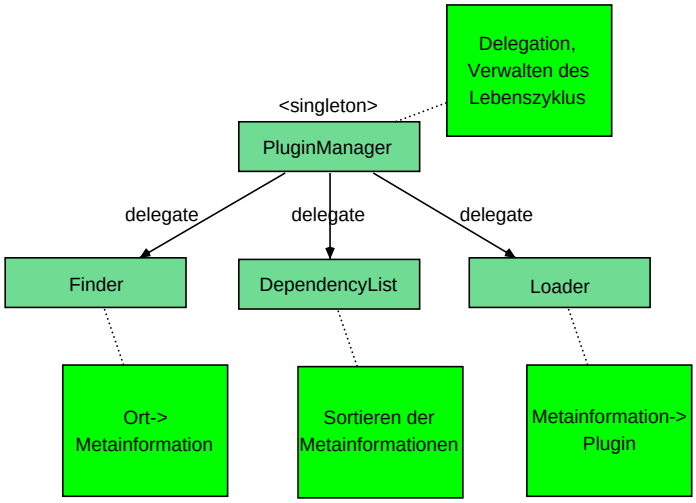


Abbildung 7.8: Aufteilung der Plugin-Verwaltung in Klassen

7.2.8 Format der Metainformationen

Wie aus der Bewertung des Prototyps ersichtlich wird, hat sich die Verwendung von XML für die Speicherung als unnötig komplex erwiesen. Insbesondere ist es im endgültigen System nicht nötig, die Metainformationen rekursiv strukturieren zu können.

Als Alternative stellt Java die Klasse `java.util.Properties` zur Verfügung, die für die Konfigurierbarkeit von Systemen gedacht und deren Dateiformat einfach zu verwenden ist. Dadurch, dass dieses bereits durch eine vorhandene Klasse gelesen werden kann, erleichtert deren Benutzung die Implementation des Systems. Außerdem ist die zu erwartende Eingewöhnung für Programmierer von *Renew*-Komponenten gering, da die Klasse aus dem Standard-Rahmenwerk des Java Development Kits, JDK, bekannt ist.

Das Format ermöglicht die Zuweisung von Schlüsseln (*keys*) an Werte (*value*), die in der Form

- `key = value` oder
- `key : value`

erfolgen kann. Auf Java-Seite wird ein `Properties`-Objekt erzeugt, das die Datei parsen kann und Methoden zur Abfrage der Werte anbietet. Die dadurch entstehende Konfiguration der Komponente sollte auch später durch die Schnittstelle abfragbar sein.

Um die im vorherigen Abschnitt geforderten Konzepte an das Pluginsystem umzusetzen, sind außerdem folgende spezielle Schlüssel von jedem Plugin anzubieten:

- **provides** und **requires** für die von der Komponente zur Verfügung gestellten respektive benötigten Dienste
- ein Eintrag **mainClass** mit dem Namen der Klasse, die als Schnittstellenobjekt instantiiert werden soll
- ein **name**-Eintrag mit einem Namen, mit dem die Komponente dem Benutzer gegenüber angezeigt werden soll.

7.3 Zusammenfassung

Dieses Kapitel behandelt den Entwurf eines Plugin-Konzeptes. In diesem liegen Dienste als Schnittstellenobjekte vor, die mit dem *Facade*-Entwurfsmuster umgesetzt werden. Eine Komponente deklariert angebotene Dienste sowie Abhängigkeiten in ihren Metainformationen, die im Format *key=value* in einer Datei vorliegt. Die Verwaltung des Lebenszyklus einer Komponente geschieht mit den Methoden `init()` und `shutdown()` aus dem Interface `IPlugin`, das alle Komponenten implementieren müssen.

Die Plugin-Verwaltung wird durch die Klasse `PluginManager` umgesetzt. Die Schritte, die zur Verwendung eines Plugins nötig sind, das *Finden*, das *sortieren* nach Abhängigkeiten sowie das *Laden* der Komponenten, delegiert sie an entsprechende Helferklassen.

Die vorgestellten Klassen liegen, in Java implementiert, vor. Die entstandene Plugin-Verwaltung ist nicht auf die Verwendung mit *Renew* beschränkt, sondern erlaubt vielmehr das Laden beliebiger Komponenten, die die Anforderungen betreffend Metainformation und Bereitstellung von Schnittstellen erfüllen. Im nächsten Kapitel wird *Renew* in Komponenten zerlegt, die von dem hier beschriebenen System geladen werden können.

Kapitel 8

Zerlegung von *Renew* in Komponenten

Ziel dieser Arbeit ist es, *Renew* verwaltbarer, konfigurierbar und erweiterbar zu gestalten. Um dies umzusetzen, ist im vorigen Kapitel eine Plugin-Verwaltung entstanden, die generische Komponenten laden und verwalten kann. Sie stellt jedoch keine fachliche Funktionalität zur Verfügung.

Nun sollen Komponenten erstellt werden, die die Funktionen von *Renew* anbieten. Dafür muss das bestehende System umgestaltet und in Form von Komponenten bereit gestellt werden. Dieses Kapitel beschreibt, wie dabei vorgegangen wird und welche Komponenten gebildet werden. Dabei werden verschiedene Möglichkeiten, das System aufzutrennen, diskutiert.

Wenn alle diese Komponenten mit ihren Metainformationen versehen und entsprechend der Spezifikation angeboten werden, kann die im vorigen Kapitel beschriebene Verwaltung sie laden. Zusammengenommen stellen sie bis auf wenige Ausnahmen wieder alle von *Renew* angebotenen Funktionen zur Verfügung.

Die Ausnahmen ergeben sich dadurch, dass an einigen Stellen Programmteile von einer Komponente in eine andere verschoben werden müssen, um die zyklischen Abhängigkeiten zu entfernen. Das nächste Kapitel beschäftigt sich damit, wie dieser Code wieder genutzt werden kann.

8.1 Analyse des bestehenden Systems

Bevor die Komponenten, in die das bestehende System aufgeteilt werden soll, festgelegt werden, wird eine Überprüfung durchgeführt, inwiefern sich das bestehende *Renew* für eine Komponentisierung eignet.

Dabei wird im Verlauf des Kapitels der Begriff der *Abhängigkeit* verwendet. Dieser wird in Abschnitt 3.1 für Komponenten definiert; hier findet sich auch die Abhängigkeit zwischen Java-*package*s:

Abhängigkeit zwischen *package*s Ein *package* `p` ist von einem *package* `o` abhängig, wenn es in `p` eine Klasse gibt, die eine Klasse aus `o` verwendet.

Da *package*s eine naheliegende Einheit sind, Komponenten zu bilden (vgl. Abschnitt 6.2.1 in [Eic02]), und die in Abbildung 6.1 dargestellte *package*-Struktur sowie die Abhängigkeiten innerhalb von *Renew* sehr übersichtlich sind, scheint eine Komponentisierung entlang von *package*-Grenzen praktikabel. Zu beachten ist dabei jedoch, dass, wie in Abschnitt 3.1 festgelegt, Komponenten nicht in gegenseitiger Abhängigkeit stehen dürfen.

Die Untersuchung der Abhängigkeiten geschieht in drei Schritten: zunächst werden diejenigen betrachtet, die in Abb. 6.1 dargestellt sind. Um die Übersichtlichkeit zu erhöhen, ist diese Abbildung auf Seite 81 noch einmal dargestellt. Da – wie bereits in Kapitel 6 erwähnt – sowohl *package*s als auch Abhängigkeiten existieren, die in Abb. 6.1 nicht erscheinen, werden diese hier vorgestellt. Dafür werden zwei getrennte Schritte durchgeführt: einer für die zusätzlichen Pakete, einer für die zusätzlichen Abhängigkeiten.

8.1.1 Untersuchung bestehender Abhängigkeitszyklen

Schon in Abbildung 6.1, die auf Seite 81 noch einmal zu sehen ist, finden sich einige gegenseitige Abhängigkeiten. Es sind jedoch wenige, und die gebildeten Zyklen enhalten nicht besonders viele *package*s.

JHotDraw

Zwei dieser Zyklen sind in dem verwendeten Grafikframework JHotDraw enthalten.

- Das *package* `CH.ifa.draw.figures` enthält Abhängigkeiten zu dem eigentlich höher angesiedelten `CH.ifa.draw.application`. Dieser Zyklus enthält vier *package*s.
- Die *package*s `CH.ifa.draw.framework` sowie `CH.ifa.draw.util` sind gegenseitig voneinander abhängig.

Diese Zyklen sind für die Komponentisierung jedoch nicht relevant. Bei JHotDraw handelt es sich um eine externe Bibliothek, auch wenn diese an einigen Stellen an *Renew* angepasst wurde.

Simulationskern

In Kapitel 6 wird *Renew* grob in die Teile *Simulationskern*, *Formalismen* und *Graphische Benutzungsoberfläche* eingeteilt. Dabei sind innerhalb des Simulationskerns zyklische Abhängigkeiten zu finden, zwischen den einzelnen Teilen jedoch nicht. Eine Aufteilung in diese drei Komponenten wäre also möglich.

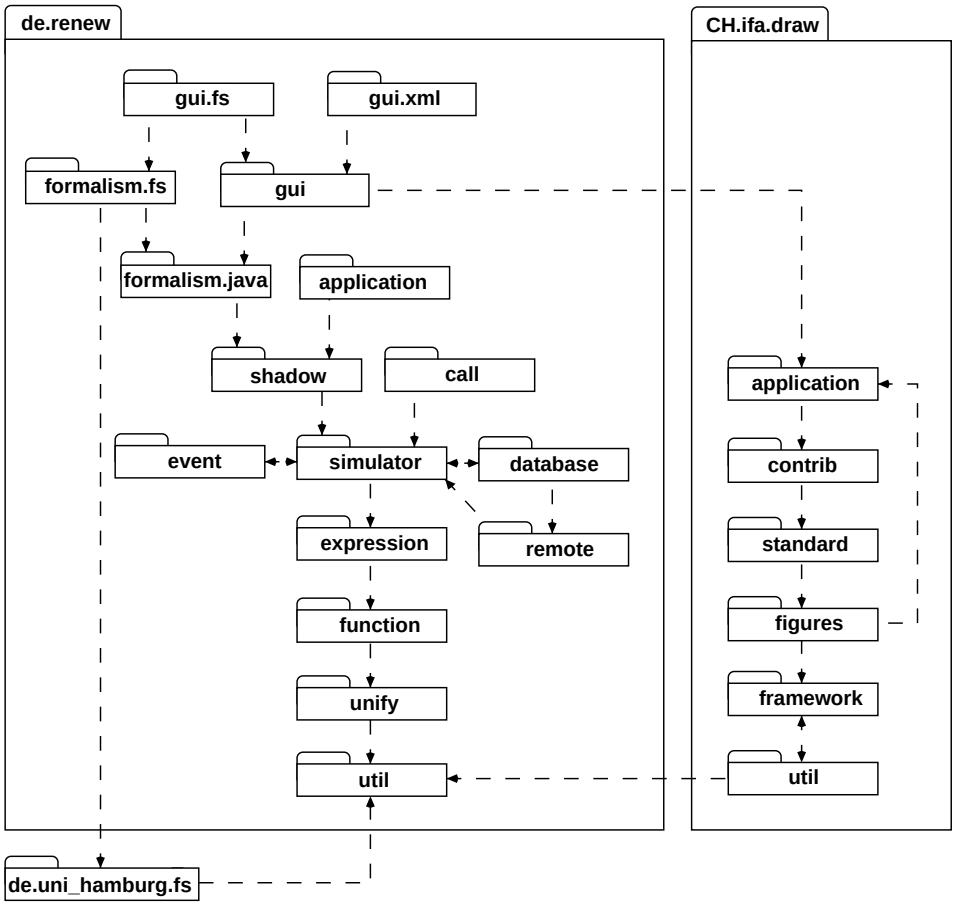


Abbildung 8.1: Die *packages* in *Renew* und ihre Abhängigkeiten (Wdh. von Abbildung 6.1)

Innerhalb des Kerns befinden sich folgende zyklische Abhängigkeiten:

de.renew.event und `de.renew.simulator` stehen in gegenseitiger Abhängigkeit.

de.renew.database und `de.renew.simulator` sind gegenseitig voneinander abhängig. Außerdem bilden die beiden *packages* mit `de.renew.remote` einen dreielementigen Zyklus.

8.1.2 Zusätzliche *packages*

In Abbildung 6.1 fehlen einige in *Renew* enthaltene *packages*, die in Abb. 8.2 zusätzlich eingetragen sind. Dabei sind die neuen *packages* heller und die durch sie entstehenden Abhängigkeiten mit gepunkteten Linien dargestellt. Der Übersichtlichkeit halber sind die *packages* `formalism.bool` und `formalism.stub` mit `f.bool` bzw. `f.stub` bezeichnet.

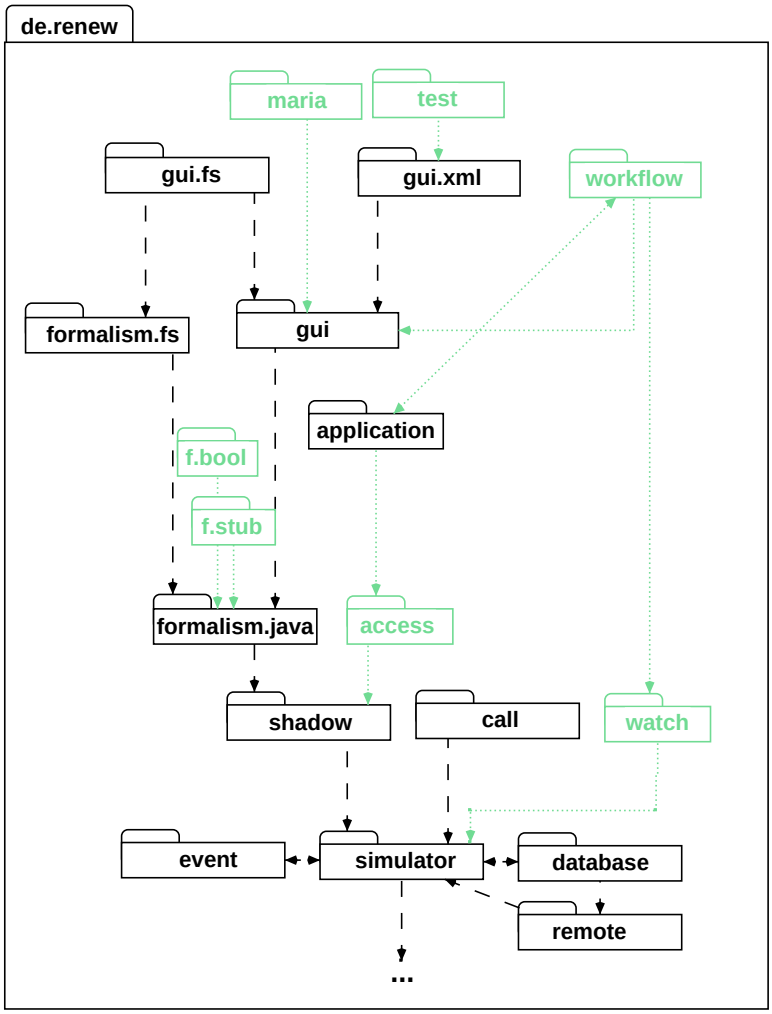


Abbildung 8.2: Zusätzliche *packages* in *Renew* (helle Rechtecke und Pfeile)

Formalismen Wie zu sehen ist, fügen die beiden Formalismus-Pakete `formalism.bool` sowie `formalism.stub` (in der Abbildung aus Platzgründen `f.bool` bzw. `f.stub`) keine neuen Zyklen in die Abhängigkeitsrelation ein. Da sie in [KWD03] bereits beschrieben werden, ist davon auszugehen, dass sie auf der Abbildung lediglich aus Gründen der Übersichtlichkeit weggelassen wurden. Das gleiche gilt für das *package* `maria` sowie für das *package* `test`. Letzteres wird zwar in der Beschreibung nicht erwähnt, es fügt sich jedoch ohne Zyklen in die vorhandene Struktur ein.

de.renew.access Dieses *package* kann dazu verwendet werden, eine Zugriffsrechteverwaltung durchzuführen. Dies geschieht über ein Framework mit Personen- und Rollen-Klassen, für die Regeln festgelegt werden können. Wie in Abb. 8.2 zu sehen, ersetzt dieses *package* die direkte Abhängigkeit zwischen `application` und `shadow`. Ein Zyklus entsteht dadurch nicht.

de.renew.watch Die Klassen in diesem *package* sind zum Überwachen von Schaltvorgängen innerhalb einer Simulation entworfen. `de.renew.watch` erzeugt keine zyklischen Abhängigkeiten.

de.renew.workflow Bei dem Paket *Workflow* handelt es sich um einen Formalismus, mit dem der Entwurf von Workflow-Netzen möglich ist. Diese Netze können während der Ausführung persistent gehalten werden.

Da für diesen Formalismus ein neues Zeichenelement, die *Task*-Transition, zur Verfügung steht, ist `de.renew.workflow` von `de.renew.gui` abhängig, damit dieses Element in die Toolbar eingefügt werden kann. Zusätzlich verwendet es zur Kontrolle der Schaltvorgänge den Mechanismus des `watch`-Paketes.

Dieses *package* ist problematisch: es erzeugt eine zyklische Abhängigkeit, die zwischen `application` und `workflow`. Diese Beziehung muss daher genauer untersucht werden; dies wird in Abschnitt 8.3 geschehen.

8.1.3 Zusätzliche Abhängigkeiten

Dieser Abschnitt beschäftigt sich mit den Abhängigkeiten zwischen den *package*s, deren Abhängigkeiten in der Abbildung 6.1 nicht vollständig eingezeichnet sind. Die fehlenden Abhängigkeiten sind in Abbildung 8.3 zu sehen. Wie im vorigen Abschnitt sind die neuen Elemente heller und durch gepunktete statt gestrichelte Linien gekennzeichnet.

de.renew.formalism.java Dieses Paket ist von `de.renew.call` abhängig. Dies ist unproblematisch, da keine zyklische Abhängigkeit entsteht.

de.renew.util Dieses *package* verweist auf Klassen aus dem Grafik-Framework JHotDraw, genauer aus dem *package* `CH.ifa.draw.framework`. Dies erzeugt eine zyklische Abhängigkeit, die von daher besonders kritisch ist, da sie ebenfalls zwischen zwei Kandidaten für die spätere Trennung besteht. Wie erwähnt, soll JHotDraw als eigenständige Komponente abgespalten werden.

CH.ifa.draw.util ist von `CH.ifa.draw.standard` abhaengig und bildet so einen Zyklus, der aber innerhalb der JHotDraw-Komponente bleibt.

de.renew.remote und `de.renew.database` sind abhängig von `de.renew.gui`. Wie schon bei `de.renew.util` entsteht hier eine zyklische Abhängigkeit, die sich über eine Grenze zweier möglicher Komponenten erstreckt, des Simulationskerns einerseits und der grafischen Benutzungsoberfläche anderseits. Der gebildete Zyklus enthält hier sechs *package*s.

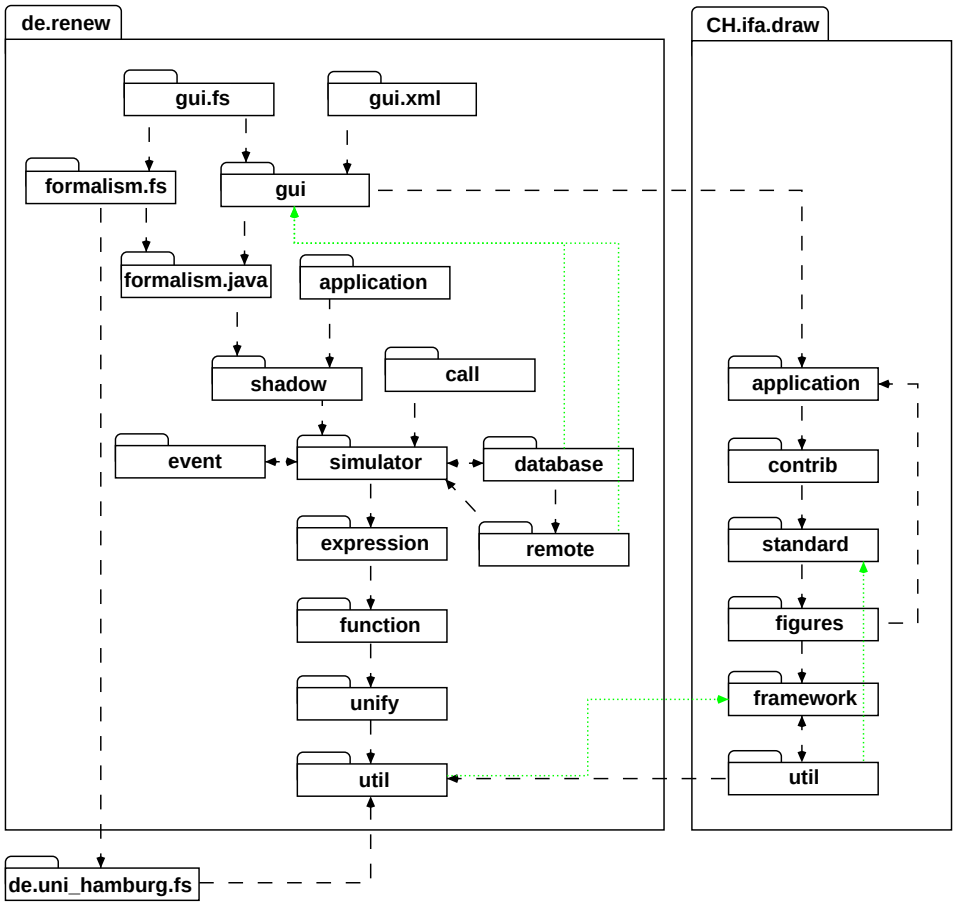


Abbildung 8.3: Zusätzliche Abhängigkeiten in *Renew* (helle Pfeile)

8.1.4 Zusammenfassung

In diesem Abschnitt wurde die *package*-Struktur des in Komponenten zu zerlegenden *Renew* auf zyklische Abhängigkeiten zwischen den *package*s untersucht, um bei der späteren Komponentisierung zu berücksichtigende Stellen zu identifizieren.

Die gefundenen Abhängigkeiten, die dabei eine Rolle spielen, sind:

- `de.renew.workflow` und `de.renew.application`
- `de.renew.util` und `CH.ifa.draw.framework`
- `de.renew.remote` und `de.renew.gui`

In Abschnitt 8.4 wird diskutiert, welche Möglichkeiten bestehen, diese Abhängigkeiten aufzulösen. Dabei werden insbesondere Entwurfsmuster [GHJV95] verwendet.

8.2 *Renew*-spezifische Anforderungen

Der vorige Abschnitt beschäftigt sich mit den technischen Randbedingungen für die Komponentisierung von *Renew*. Auf der anderen Seite ist es jedoch auch notwendig, fachliche Aspekte zu berücksichtigen; letztendlich soll es sich bei den Komponenten um fachliche Einheiten handeln.

8.2.1 Grafische Benutzungsoberfläche

Wie schon vorher erwähnt, ist die grafische Benutzeroberfläche ein integraler Bestandteil von *Renew*. Während sie aber zum Erstellen und Anzeigen der Netze geeignet ist, wird sie für deren Simulation nicht benötigt.

In der Tat kann schon im bisherigen System ein *Renew*-Server gestartet werden, über den auf eine laufende Simulation zugegriffen werden kann. Auch die Möglichkeit, eine Simulation ohne eine GUI starten zu können, um Ressourcen auf dem ausführenden Rechner zu sparen, existiert bereits. Wie das geht, ist jedoch erstens in der aktuellen Version von *Renew* nicht dokumentiert, und zweitens müssen durch den monolitischen Aufbau des Programms auch dann alle Klassen ausgeliefert werden, wenn keine graphische Unterstützung benötigt wird.

Ebenfalls im Zusammenhang mit der grafischen Benutzung von *Renew* ist noch das JHotDraw-Framework zu erwähnen, das für die Erstellung der GUI verwendet wird. Da es sich hierbei um eine externe Bibliothek handelt, ist eine Abspaltung dieser *package*s vom Restsystem nicht nur wegen der darin vorkommenden Zyklen (vgl. Abschnitt 8.1.1), sondern auch unter fachlichen Gesichtspunkten sinnvoll.

8.2.2 Konfigurierbarkeit des Compilers

Im nicht komponentisierten *Renew* ist die wichtigste Konfigurationsmöglichkeit die Auswahl des *Mode*s. Dieser bestimmt, welcher Formalismus verwendet wird; dies ist gleichbedeutend damit, welcher Compiler zum Einsatz kommt. Dabei muss der *Mode* beim Starten des Programmes anhand einer Kommandozeilenoption oder durch Setzen einer Umgebungsvariable festgelegt werden; ein nachträgliches Ändern des verwendeten Formalismus ist nicht möglich, ohne das Programm neu zu starten.

Damit ist eine weitere wünschenswerte Eigenschaft identifiziert, die *Renew* bieten sollte: ein Wechsel des Formalismus zur Laufzeit. Dies ist eine Vorbereitung auf eine weiter führende Idee, auf die im Rahmen der Entwicklung von *Renew* hingearbeitet werden soll. Dabei ist das Ziel, für jedes gezeichnete Netz einen anderen Compiler verwenden zu können.

Zum anderen enthält der *Mode* auch Informationen, mit denen bestimmte Zeichenelemente, die nur für diesen Formalismus sinnvoll sind, der GUI zur Verfügung gestellt werden. Der Simulator verwendet den *Mode* nur, wenn er mit der graphischen Oberfläche gestartet wird; anderenfalls bezieht er den Formalismus aus dem zu simulierenden Netz, das in einer Datei gespeichert vorliegt. Mit der Einführung von Komponenten sollen diese beiden Arten, den Simulator zu konfigurieren, vereinheitlicht werden.

8.3 Aufteilung in Komponenten

Nachdem die Randbedingungen der Komponentisierung bestimmt sind, wird in diesem Abschnitt nun die Aufteilung von *Renew* erarbeitet. Dazu werden verschiedene Möglichkeiten, die sich aus der vorherigen Diskussion ergeben, analysiert und beurteilt. Am Ende steht dann eine Komponentenstruktur, die sowohl die technischen als auch die fachlichen Bedingungen erfüllt.

8.3.1 Zusammenfassen der Zyklen

Die in Abschnitt 8.1 beschriebenen technischen Gegebenheiten liegen dem ersten Vorschlag zu Grunde. Die dort identifizierten Abhängigkeiten zwischen den *package*s bieten eine erste Möglichkeit, das System zu gliedern.

Da sich zwischen den entstehenden Komponenten keine Zyklen bilden dürfen, können Komponenten einfach gebildet werden, indem die vorhandenen Zyklen in einzelne Teile zusammenzufasst werden. Eine solche Einteilung ist in Abbildung 8.4 dargestellt.

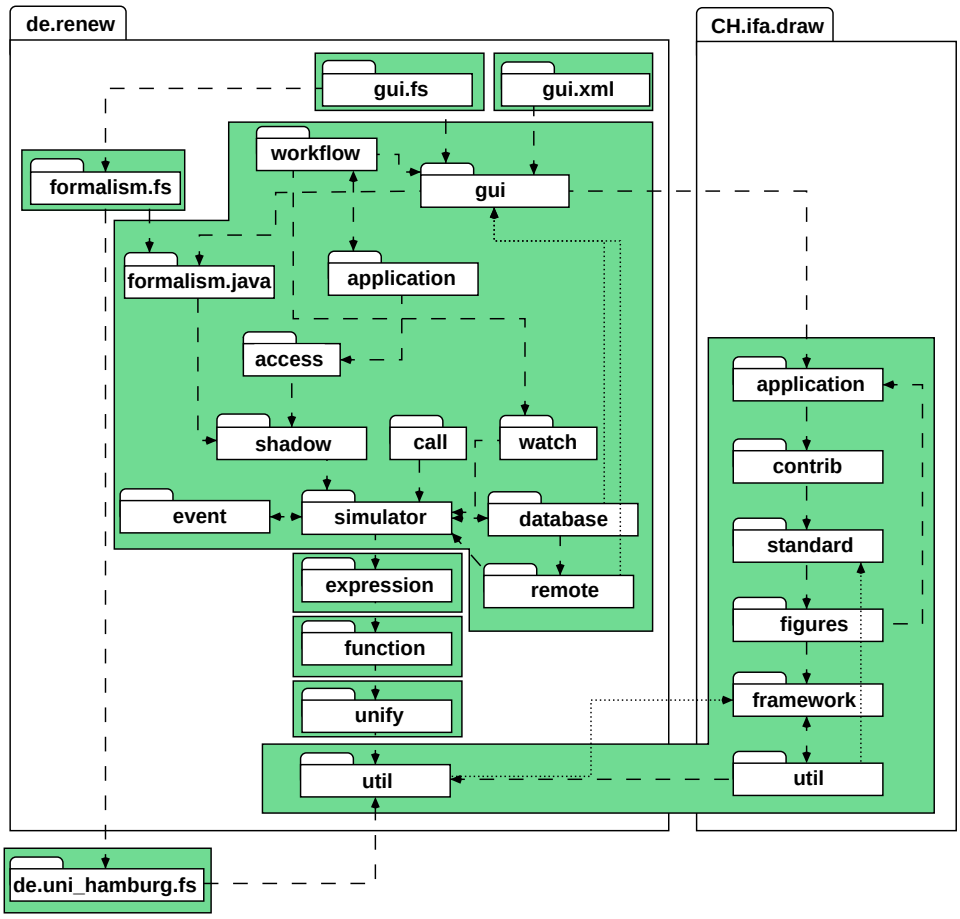


Abbildung 8.4: Komponentisierung nach technischen Gesichtspunkten (Zusammenfassung von Zyklen)

Die gewählten Komponenten decken minimal die Abhängigkeitszyklen zwischen den *package*s ab. Bei dieser Aufteilung ist auffällig, dass sich zwar viele kleine Komponenten bilden, die nur einzelne *package*s enthalten (etwa `de.renew.expression` oder `de.renew.formalism`), der Großteil der Pakete sich jedoch auf zwei Komponenten aufteilt. Ein solches Mißverhältnis ist ein Zeichen für zu große Zyklen im Ursprungssystem, und auch für ein Komponentensystem sind die Unterschiede in der Granularität zwischen den einzelnen entstandenen Komponenten nicht wünschenswert.

Zudem ergeben sich ungewünschte fachliche Zusammenfassungen, etwa der Zugehörigkeit von `de.renew.util` zu den JHotDraw-Pakten oder der Zusammenfassung von der grafischen Oberfläche mit dem Simulationskern. Diese Punkte werden im Folgenden besprochen.

8.3.2 Aufteilung in fachliche Teile

Die Zusammenfassung von Zyklen ist zwar eine Möglichkeit, *Renew* zu gliedern, entspricht aber nicht dem Sinn von Komponenten, der ursprünglich die Wiederverwendung von einzelnen, semantischen Teilen zum Ziel hat. Daher sollten insbesondere fachliche Aspekte die Aufteilung bestimmen.

Als grobe Bestandteile von *Renew* werden bereits in Kapitel 6 drei Hauptgebiete festgestellt:

- Simulationskern
- Formalismen
- GUI

Der Punkt *Formalismen* passt zu der in Abschnitt 8.2.2 ausgedrückten Forderung, den zu verwendenden Compiler zur Laufzeit wechseln zu können. Als Ansatz ergibt sich daraus die Vorstellung einer Komponente, die die zur Verfügung stehenden Formalismen speichert, und bei der der Benutzer einen Compiler auswählen kann, der dann die anschließenden Übersetzungen übernimmt.

Die Anforderung, die in Abschnitt 8.2.1 zu finden ist, betrifft die Behandlung von JHotDraw: dieses *package* soll eine eigenständige Komponente ergeben, also nicht in die zu erstellende **Gui**-Komponente eingefügt werden.

Die Aufteilung, die sich unter Berücksichtigung dieser Punkte ergibt, ist auf Abbildung 8.5 zu sehen. Zur Verdeutlichung sind die drei getrennten *Formalismen* in der linken oberen Ecke durch einen Rahmen zusammengefasst. Auf dieser Darstellung sind nur noch Abhängigkeiten zwischen den Komponenten abgebildet.

Die Zyklen, die sich in der Abhängigkeitsrelation bilden, sind

- **Simulator** und **Gui**
- **Simulator** und **JHotDraw**
- **Gui** und **formalism.fs**

Diese Abhängigkeiten sind in Abbildung 8.5 als hellere, gepunktete Linien dargestellt.

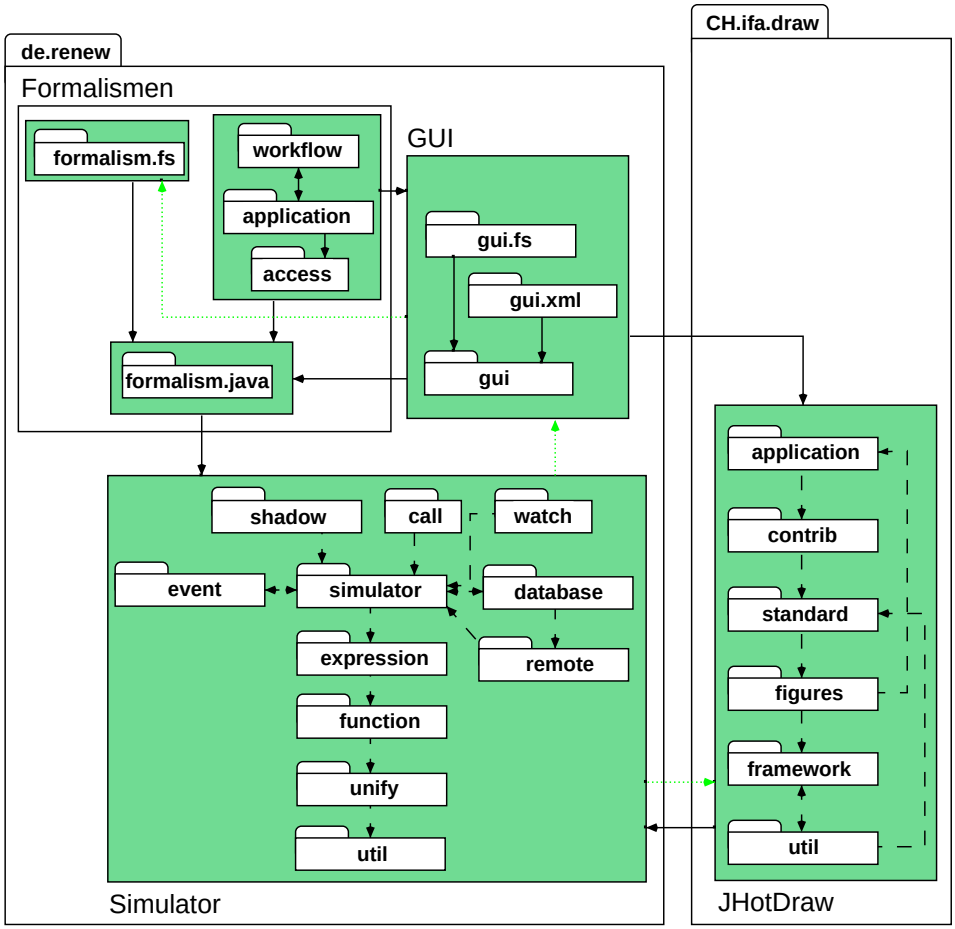


Abbildung 8.5: Komponentisierung nach fachlichen Gesichtspunkten

8.3.3 Verfeinerung

Insgesamt ist die aus Abschnitt 8.3.2 resultierende Aufteilung bereits ein guter Ansatz für die Komponentisierung von *Renew*. Bevor die notwendigen Schritte zur Beseitigung der zyklischen Abhängigkeiten in Angriff genommen werden, sollte jedoch die endgültige Aufteilung feststehen.

Die in Abb. 8.5 dargestellte Komponentenstruktur hat den Nachteil, dass die **Simulator**-Komponente unverhältnismäßig viele Klassen enthält. Außerdem gehört das *package* `de.renew.gui.fs` eher in das Paket `formalism.fs` als in **Gui**; es dorthin zu verschieben ist jedoch einfach und löst zudem noch einen der beschriebenen Abhängigkeitszyklen. Um die **Simulator**-Komponente weiter aufzutrennen, können wie in Abschnitt 8.3.1 beschrieben zyklische Abhängigkeiten zusammengefasst werden.

Unter diesem Gesichtspunkt ist ein Kandidat zum Zusammenfassen der Bereich um das *package* `de.renew.simulator`. Dieser umfasst ebenfalls `de.renew.event`, `de.renew.database` sowie `de.renew.remote`. In den Klassen aus diesen Paketen findet sich die Funktionalität zum Simulieren der Netze. Da das *package* `de.renew.shadow` für die Datenstruktur der zu simulierenden Netze verwendet wird, sollte es ebenfalls mit eingebracht werden.

Die Simulatorklassen basieren sie auf den darunter liegenden *package*s, also `expression`, `function`, `unify` sowie `util`. Die ersten drei wiederum bilden einen semantischen Komplex, der, wie in Kapitel 6 beschrieben, mit dem Binden von Variablen an Werte sowie dem Auswerten von den den Netzen zugehörigen Inschriften befasst ist. Dies wird dadurch bestätigt, dass die Klassen aus einem dieser *package*s selten verwendet werden, ohne Klassen aus einem der anderen beiden ebenfalls zu benutzen. Dies legt die Bildung einer Komponente aus diesen *package*s nahe.

Das *package* `util` hingegen verfügt über sehr vielseitige Klassen, so dass es gerechtfertigt ist, eine eigenständige Komponente daraus zu erstellen. Das Gleiche gilt für das *package* `de.renew.access`.

Die übrig gebliebenen *package*s `watch` und `call` erhalten der Vollständigkeit halber ebenfalls eine Komponente **Misc** für *Verschiedenes*, da sie sich fachlich nicht den anderen Bestandteilen zuordnen lassen.

Um den Bearbeitungsaufwand für die anliegenden Umstrukturierungen gering zu halten, werden als letzter Schritt vor Entfernung der Abhängigkeitszyklen zwischen den Komponenten noch die Formalismen `formalism.bool` und `formalism.java` in einer Komponente zusammengefasst. Es muss jedoch dennoch möglich sein, neue Formalismen in das System einzubringen, schon um die beiden anderen Formalismen *Feature Structures* sowie *Workflow* auch nach dem Umbau noch einsetzen zu können.

Das Ergebnis der beschriebenen Korrekturen ist in Abbildung 8.6 dargestellt. Die noch vorhandenen zyklischen Abhängigkeiten sind wieder durch gepunktete helle Pfeile zwischen den Komponenten gekennzeichnet.

8.3.4 Auftrennen der *Mode*s

Der Compiler ist in der nicht erweiterbaren Fassung von *Renew* durch ein so genanntes *RenewMode*-Objekt vorgegeben. Hierbei handelt es sich um ein Interface, das im *package* `de.renew.gui` deklariert wird. Die konkrete Klasse, die das System verwenden soll, kann vom Benutzer beim Systemstart festgelegt werden.

Der Grund, warum sich das Interface in `de.renew.gui` befindet, liegt darin, dass vom Mode auch die Darstellung bestimmter grafischer Elemente beim Ablauf der Simulation abhängt. Zusätzlich kann der gewählte *Mode* Werkzeuge zum Zeichnen neuer Elemente

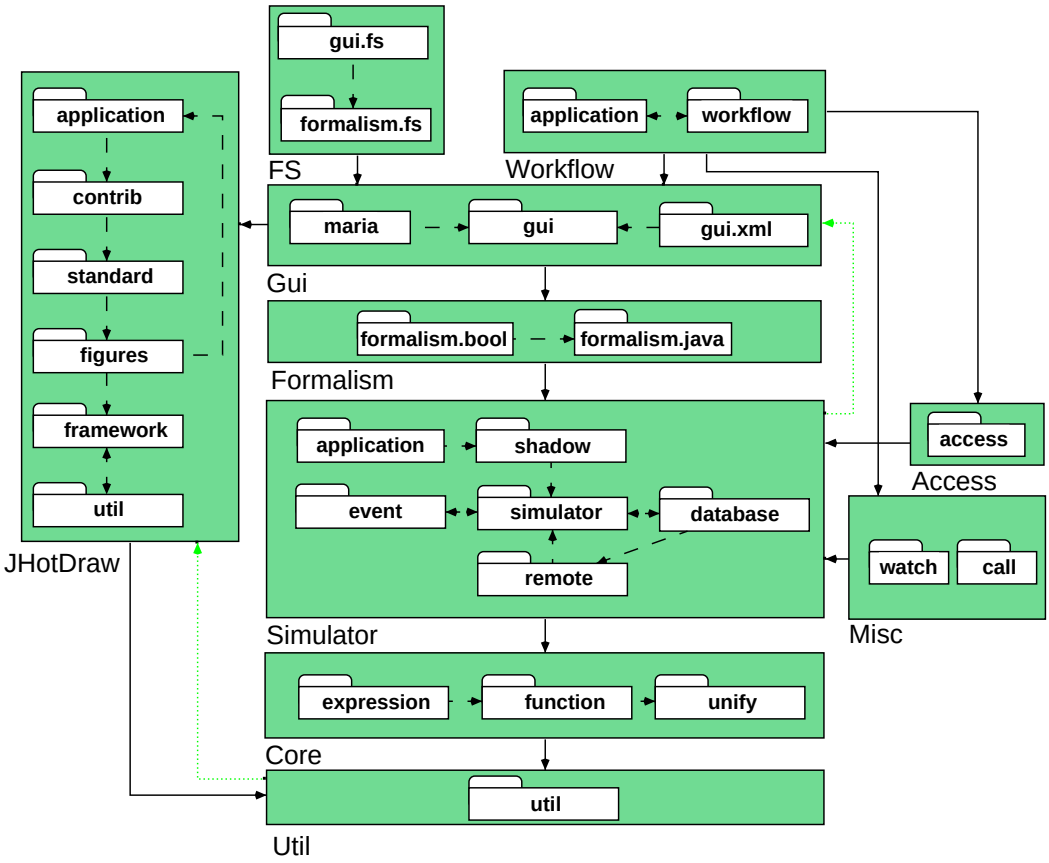


Abbildung 8.6: Endgültige Komponentisierung von *Renew* mit zu beseitigenden Abhängigkeiten (helle Pfeile)

in die Gui einbringen. Um die Trennung der beiden Komponenten **Gui** und **Formalism** vornehmen zu können, müssen diese beiden Funktionalitäten voneinander getrennt werden.

Dabei ist es fachlich zwingend, dass sich der Compilerteil im Formalism-Plugin befindet. Nur für die Darstellung der Elemente ist ein Umbau notwendig.

Gui stellt alle Elemente dar Die erste Möglichkeit besteht darin, dass alle vorhandenen Modes so zusammengefasst werden, dass eine Darstellungsklasse entsteht. Diese kann dann eine Anzeige für alle auftretenden Fälle erzeugen.

Das Problem an diesem Ansatz ist es, dass auch der *Feature Structure*-Mode eine Darstellung für Elemente bietet. Da die entsprechende Komponente aber von **Gui** abhängig ist, entstünde ein Abhängigkeitszyklus, wenn letztere eine Klasse des FS-Plugins verwenden würde. Außerdem beschränkt diese Lösung die Erweiterbarkeit des Systems, da es hierdurch unmöglich wird, neue Anzeigeklassen in das System einzubringen.

Es folgt eine Aufzählung der Möglichkeiten, wie die Darstellung von Netzelementen zur Simulation bei einer Auftrennung des *Mode*s behandelt wird.

Nur Standarddarstellung erlauben Die zweite Möglichkeit besteht darin, lediglich einfache Darstellungen zu erlauben. Diese müssten ebenfalls von Klassen in der Gui-Komponente erzeugt werden, da übergeordnete nicht verwendet werden können und untergeordnete die grafischen Funktionen nicht kennen.

Diese Möglichkeit würde jedoch dazu führen, dass Funktionalität, die vor der Komponentisierung vom System bereit gestellt wurde, danach nicht mehr angeboten würde. Da die Aufteilung in Komponenten jedoch erklärterweise keine Einschränkung von *Renew* bedeuten soll, ist auch dies nicht annehmbar.

Gui ist um Darstellung erweiterbar Der letzte Weg zu einer Aufteilung des Modes ist es, der Gui eine Standardfunktion für die Darstellung zu geben, es jedoch über ihre Schnittstelle zu ermöglichen, ihr andere Anzeigeobjekte anzubieten.

Dies ist der flexibelste Weg, kann jedoch erst beim Entwurf der Schnittstellen der Komponenten umgesetzt werden. Es handelt sich jedoch um eine gangbare Möglichkeit und soll daher eingesetzt werden.

Analog gilt für zusätzliche Werkzeuge, die die jetztigen *Mode*s zur Verfügung stellen, dass die **Gui** eine entsprechende Erweiterungsschnittstelle anbieten sollte.

Die Beschreibung der Schnittstellen, die die Erweiterung der Gui ermöglichen, erfolgt in Kapitel 9. Bis diese fertig gestellt sind, erzeugt die Gui lediglich eine Standarddarstellung wie oben beschrieben.

8.4 Entfernen der zyklischen Abhängigkeiten

Nachdem die Komponenten, in die *Renew* aufgeteilt werden soll, festgelegt sind, müssen die noch vorhandenen zyklischen Abhängigkeiten, die in Abb. 8.6 zu sehen sind, beseitigt werden.

Dieses Kapitel beschreibt einige Vorgehensweisen, die zur Lösung dieser Aufgabe angewendet werden können. Die Wahl der Methode ist jeweils davon abhängig, wodurch die Abhängigkeit entsteht. Daher wird für jede Herangehensweise zunächst beschrieben, in welchem Fall sie anzuwenden ist, bevor die durchzuführenden Schritte aufgelistet werden. Dieses Vorgehen verwendet auch Fowler [Fow99].

8.4.1 Identifizieren der abhängigen Klassen

Zur Auflösung der zyklischen Abhängigkeit zwischen Paketen ist es notwendig, zu identifizieren, wodurch die Abhängigkeit entsteht.

Im Falle von Java lässt sich dies dadurch identifizieren, dass eine Klasse, die eine andere verwendet, dies mit dem Schlüsselwort `import` deklariert[1].

Ein dafür entwickeltes UNIX-Shellscript sucht nach eben diesen `import`s und fasst sie für alle *package*s zusammen. Einen großer Teil des Scripts besteht aus diesem Zusammenfassen; die Abhängigkeiten zwischen einzelnen Klassen zu finden ist wesentlich weniger aufwändig. Mit dem Befehl

```
grep "^import some\.package" ‘find $DIR -name "*.java"‘
```

etwa werden die Java-Dateien ausgegeben, die `some.package` importieren; dabei muss die Variable DIR das zu durchsuchende Verzeichnis enthalten.

Util und JHotDraw

Wie in Abb. 8.6 zu sehen ist, gibt es in *Renew* nach der Auftrennung zwei Abhängigkeiten, für die eine Analyse erfolgen muss. Dies ist zum einen diejenige zwischen den Komponenten **Util** und **JHotDraw**. Für diesen Fall lautet das oben angegebene Kommando

```
grep "^import CH" ‘find src/de/renew/util -name "*.java"‘
```

Aus diesem Aufruf ergeben sich zwei Java-Dateien aus `de.renew.util`, die von Klassen aus **JHotDraw** abhängig sind. Dabei handelt es sich um `de.renew.util.GraphLayout` sowie `de.renew.util.PostscriptWriter`. Eine genauere Untersuchung, wodurch diese Abhängigkeit zu Stande kommt und wie sie aufzulösen ist, folgt in Abschnitt 8.4.2.

Simulator und Gui

Die andere problematische Abhängigkeit in Abbildung 8.6 ist die von **Simulator** nach **Gui**. Da die **Simulator**-Komponente mehrere *package*s enthält, ist es nötig, jedes der

[1] Statt des `import`s kann auch bei jeder Benutzung einer Klasse ihr kanonischer Name angegeben werden. Dies ist jedoch nicht gängig und kann leicht in die oben angegebene Form umgewandelt werden. Die andere Möglichkeit, wie die Benutzung zwischen Klassen mit der beschriebenen Methode unentdeckt bleiben kann, ist, wenn der betreffende Code die Java Reflection API verwendet. Dies ist im Einzelfall nachzuprüfen.

entsprechenden Verzeichnisse mit dem oben angegebenen Befehl nach abhängigen Klassen zu durchsuchen.

Dabei kommen folgende Beziehungen zu Tage:

- `de.renew.database.SetupHelper` importiert die Klasse `CPNSimulation` aus dem *package* `de.renew.gui`
- `de.renew.remote.ObjectAccessor` sowie `de.renew.remote.ObjectAccessorImpl` importieren `de.renew.gui.RenewMode`.

Die Arten der Abhängigkeiten, die in diesem Abschnitt beschrieben werden, können sicherlich noch um andere ergänzt werden; es handelt sich um die Fälle, die in *Renew* auftreten. Diese wurden abstrahiert, damit die beschriebene Vorgehensweise allgemeiner angewendet werden kann. Es sind jedoch sicher Situationen denkbar, für die andere Verfahren zur Restrukturierung notwendig sind.

8.4.2 Verschieben von Klassen zwischen Komponenten

Dies ist der einfachste Fall für die Auflösung von zyklischen Abhängigkeiten. Es handelt sich dabei um ein Refactoring, das Fowler [Fow99] nicht behandelt, aber im Online-Katalog [Fow03] nachgeliefert wird.

Anwendbarkeit

Er tritt auf, wenn eine einzelne Klasse `k` von einer Klasse `t` aus einem übergeordneten *package* `oberhalb` abhängig ist, sie selbst jedoch nicht aus einem *package* verwendet wird, von dem `oberhalb` abhängig ist. Dieser Fall ist in Abbildung 8.7 dargestellt.

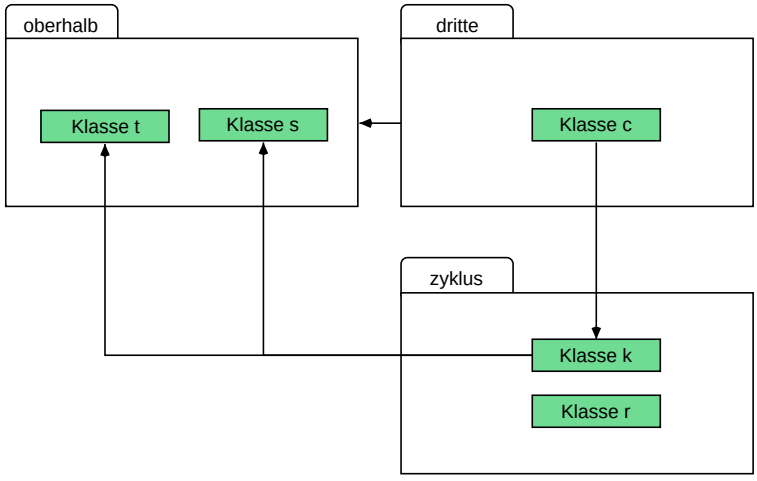


Abbildung 8.7: Bedingung zur Anwendbarkeit von *Klasse zwischen Komponenten verschieben*

In diesem Fall enthält das *package* `zyklus` die problematische Klasse `k`, die von dem übergeordneten *package* `oberhalb` abhängig ist. Zu beachten ist, dass alle Klassen, die `k` verwenden (auf der Abbildung z.B. `c`), sich nicht in Paketen unter `oberhalb` befinden. Wäre dies der Fall, entstünde durch das Verschieben der Klasse ein neuer Zyklus.

Vorgehen

Das Entfernen des Abhängigkeitszyklus in diesem Fall lässt sich durch das Verschieben der Klasse `k` in das übergeordnete *package* `oberhalb` bewältigen. Im Fall des oben angeführten Beispiels wäre das Ergebnis dieser Verschiebung eine Struktur wie in Abbildung 8.8.

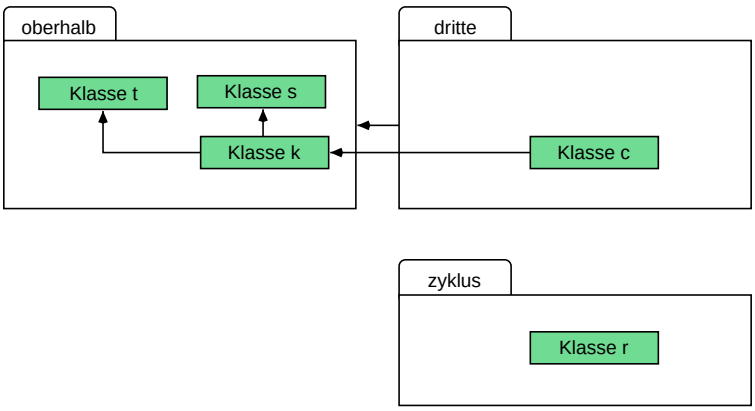


Abbildung 8.8: Ergebnis des Verschiebens der Klasse mit zyklischer Abhängigkeit

Die für diese Operation auszuführenden Schritte sind:

1. Verschieben der Quelldatei der Klasse `k` in das entsprechende Verzeichnis des *package*s `oberhalb`
2. Korrigieren der *package* -Statements in der Quelldatei der Klasse `k`
3. Korrigieren der `import`-Statements in den von `k` abhängigen Klassen (im Beispiel die Klasse `c`)

Beispiel aus *Renew*

Auch bei *Renew* taucht dieser einfache Fall auf. Bei den zu verschiebenden Klassen handelt es sich um diejenigen, die im *package* `util` als problematisch identifiziert wurde: `de.renew.util.GraphLayout` und `de.renew.util.PostscriptWriter`.

Eine Überprüfung der Klasse `de.renew.util.GraphLayout` zeigt, dass die einzigen Klassen, die diese benutzen, im *package* `de.renew.gui` liegen (genauer handelt es sich um `de.renew.gui.CPNDrawing` und `de.renew.gui.LayoutFrame`). **Gui** ist bereits von **JHotDraw** abhängig. Damit sind die oben postulierten Bedingungen erfüllt und `de.renew.util.GraphLayout` kann nach `CH.ifa.draw.util` verschoben werden.

Im Falle von `de.renew.util.PostscriptWriter` gilt ähnliches. Lediglich `de.renew.gui.CPNApplication` aus `Gui` und `CH.ifa.draw.figures.PolyLineFigure` aus `JHotDraw` verwenden diese Klasse. Damit steht auch hier einer Verschiebung nach `CH.ifa.draw.util` nichts im Wege.

8.4.3 Ersetzen einer Klasse durch ein Interface

Dieser Abschnitt beschreibt, wie zwei Pakete dadurch getrennt werden können, dass eine Klasse durch ein neu eingeführtes Interface ersetzt wird. Diese Umstrukturierung ist als *Extract Interface* [Fow99, S. 341] bekannt. In diesem Fall ist nicht allein die statische Struktur zwischen den Klassen maßgeblich, sondern auch die Art und Weise, wie die Aufrufe der Methoden, die die Abhängigkeit verursachen, konkret aussehen.

Anwendbarkeit

Diese Methode lässt sich anwenden, wenn eine Klasse `l` aus einem untergeordneten *package* `unterhalb` von einer Klasse `k` aus einem übergeordneten *package* `oberhalb` abhängig ist und `l` nicht nach `oberhalb` verschoben werden kann, weil dieses durch von `l` abhängigen Klassen einen Zyklus in die Abhängigkeit bringen würde.

Dabei sollten sich die Methoden, die auf `k` aufgerufen werden, fachlich sinnvoll zusammenfassen lassen und keine Argumente mit in `oberhalb` deklarierten Typen enthalten. Außerdem sollte die Klasse `l` keine Exemplare von `k` erzeugen, sondern diese von aussen erhalten.

Abb. 8.9 zeigt die allgemeine Struktur dieser Situation.

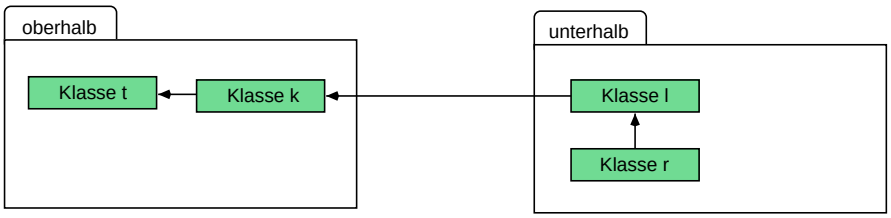


Abbildung 8.9: Ausgangsstruktur zur Anwendbarkeit von *Erzeugen eines Interfaces*

Vorgehen

Um diesen Abhängigkeitszyklus aufzulösen, werden folgende Schritte ausgeführt:

1. Die in `l` aufgerufenen Methoden werden in einem neuen Java-Interface `i` in *package* `unterhalb` gesammelt.

2. Die Deklaration der Klasse `k` wird so erweitert, dass sie `i` implementiert.

3. Alle Methoden, die bei `l` ein `k`-Objekt als Parameter enthalten, werden so umdeklariert, dass sie ein `i`-Objekt erhalten.

4. Es könnte notwendig sein, dass `l` ein Exemplar von `k` von einer Klasse aus `oberhalb` übergeben bekommt.

Als Ergebnis steht am Ende dieser Transformation eine Struktur wie auf Abbildung 8.10.

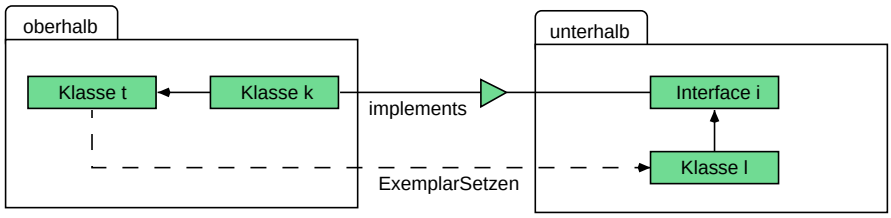


Abbildung 8.10: Struktur nach Anwendung *Erzeugen eines Interfaces*

Beispiel aus *Renew*

Dieser Fall trifft für den ersten Punkt der beiden Abhängigkeiten zwischen den Komponenten **Gui** und **Simulator** (Seite 93). Er betrifft die Klassen `de.renew.database.SetupHelper` sowie `de.renew.gui.CPNSimulation`.

Der fachliche Sinn hinter dieser Beziehung ist, dass der `SetupHelper` aus `de.renew.database` mit der Methode `setup` den in einer Datenbank gespeicherten Zustand einer vorher gelaufenen Simulation wiederherstellt.

`de.renew.gui.CPNApplication` ruft diese Methode auf, wenn die grafische Oberfläche die Simulation initiiert. `de.renew.application.ShadowSimulator` tut dies, wenn eine Server-Version der Simulation laufen soll. Letzteres übergibt aber lediglich `null` als Argument, erzeugt also kein Exemplar der Klasse.

Analog zu der oben angegebenen Vorgehensweise werden die folgenden Schritte unternommen:

1. Die in `de.renew.database.SetupHelper` aufgerufenen Methoden werden im neuen Java-Interface `de.renew.database.Simulation` gesammelt

2. Die Deklaration der Klasse `de.renew.gui.CPNSimulation` wird so erweitert, dass sie `de.renew.database.Simulation` implementiert

3. Die statische Methode `setup()` in `SetupHelper`, die `de.renew.gui.CPNSimulation` als Parameter hat, wird so umdeklariert, dass sie ein `Simulation`-Objekt erhält

Erweiterung Wie oben beschrieben, eignet sich diese Methode besonders, wenn die Klasse `l` kein Exemplar der Klasse `k` erzeugt. Im genannten Beispiel ist lediglich eine statische Methode aus `l`, die `k` als Parameter übernimmt, betroffen.

Eine Möglichkeit, diese Trennung auch dann vorzunehmen, wenn `k` von `l` instanziiert wird, besteht in der Verwendung des Entwurfsmusters *Abstract Factory*.

Die Fabrik-Klasse wird so deklariert, dass sie eine Methode `createInstance()` enthält, die ein Exemplar von `k` zurückliefert. Ein Aufruf dieser Methode wird an die Stellen gesetzt, an denen `l` vorher ein Objekt von `k` erzeugt hat. In dem übergeordneten *package* `oberhalb` muss in diesem Fall dafür Sorge getragen werden, dass `l` ein entsprechendes Factory-Objekt erhält. Außerdem muss `l` darauf vorbereitet sein, dass keine Factory zur Verfügung steht, da das System so konfiguriert sein kann, dass das übergeordnete Paket nicht geladen wird.

8.4.4 Ändern der Methodenparameter und Aufrufreihenfolge

Dieser Abschnitt behandelt den Fall, dass die ungewollte Abhängigkeit durch eine bestimmte Sequenz von Methodenaufrufen entsteht. Dabei muss eine Klasse einen Aufruf an eine Methode aus einem übergeordneten *package* delegieren.

Da dieser Fall die Delegationsbeziehungen zwischen Klassen behandelt, ist er mit dem Refactoring *Remove Middle Man* [Fow99, S. 160] verwandt.

Anwendbarkeit

Dieses Vorgehen ist anwendbar, wenn eine Klasse `k` aus einem *package* `oben` eine Methode aus einer Klasse `l` aus einem untergeordneten Paket `unten` aufruft und diese den Aufruf an eine dritte Klasse `m` delegiert. Dabei befindet sich `m` im *package* `mitte`, das in der Abhängigkeitsrelation oberhalb von `unten`, jedoch unterhalb von `oben` steht.

Abbildung 8.11 veranschaulicht die genannten Abhängigkeiten. Die Reihenfolge der Aufrufe, die für den hier beschriebenen Umbau nötig ist, ist auf Abbildung 8.12 zu sehen. Das Problem wird verursacht, indem die Klasse `l` eine Methode aus der Klasse `m`, die sich in einem höheren Paket befindet, aufruft.

Vorgehen

Zur Beseitigung der Abhängigkeit zwischen `m` und `l` wird der Aufruf der Methode, den `k` an `l` richtet, durch einen Aufruf von `k` nach `m` ersetzt. Dies erzeugt keinen neuen Abhängigkeitszyklus, da das Paket `oben` von `k` nach Vorraussetzung von *package* `w`, in dem sich `m` befindet, abhängig ist. Die Daten, die zur Ausführung der neuen Methode nötig sind, holt sich `m`, indem es `l` aufruft, das wiederum in einem untergeordneten Paket deklariert ist.

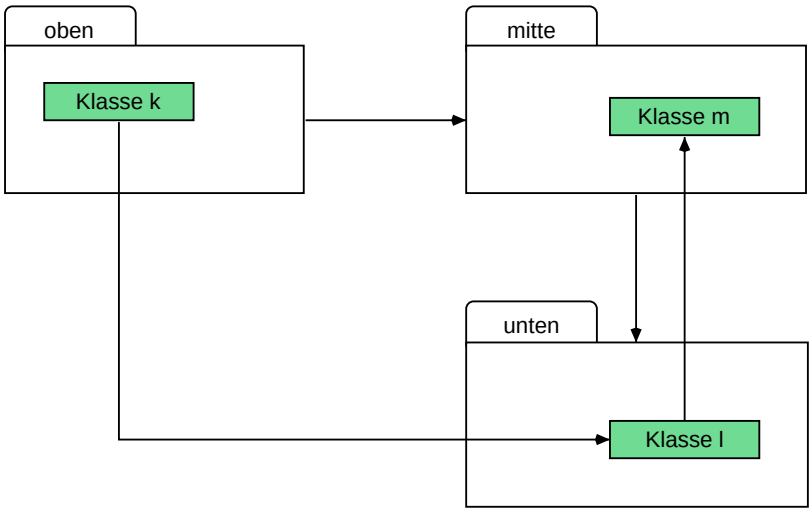


Abbildung 8.11: Abhängigkeiten zwischen Paketen und Klassen, wie für die Durchführung der Methode in Abschnitt 8.4.4 nötig ist

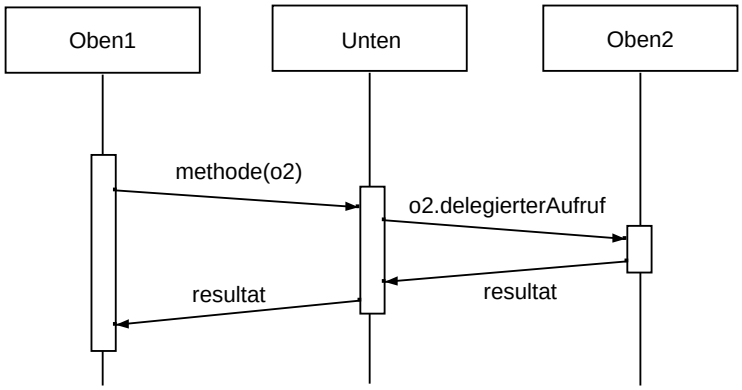


Abbildung 8.12: Aufrufsequenz, die zyklische Abhängigkeit erzeugt: `delegierterAufruf` geht an ein Objekt aus einem übergeordneten *package*

Das Ergebnis sind die in Abbildung 8.14 dargestellten Abhängigkeiten. Vielsagender ist jedoch die Reihenfolge der Methodenaufrufe, wie sie in Abb. 8.13 als Sequenzdiagramm zu sehen sind.

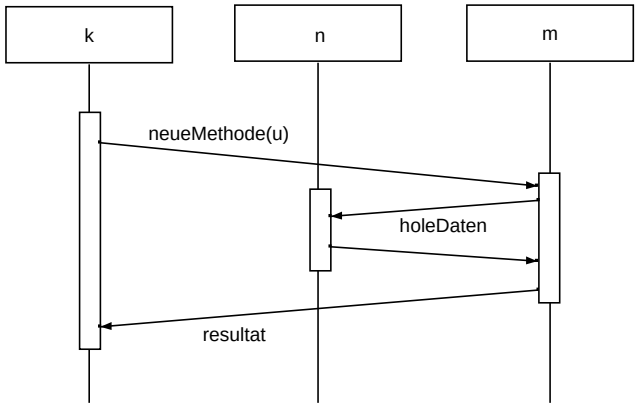


Abbildung 8.13: Sequenzdiagramm der Methodenaufrufe nach Umstellung gemäß Abschnitt 8.4.4

Beispiel aus *Renew*

Die Umstrukturierung von Methodenaufrufen beseitigt die letzte auf Seite 93 beschriebene Abhängigkeit, die zwischen `de.renew.remote.ObjectAccessorImpl` und `de.renew.gui.RenewMode` besteht.

Bei den Klassen `k`, `l` und `m` handelt es sich in diesem Fall um die Klassen `MultipleTokenFigure`, `ObjectAccessorImpl` sowie `RenewMode` aus den *packages* `de.renew.gui`,

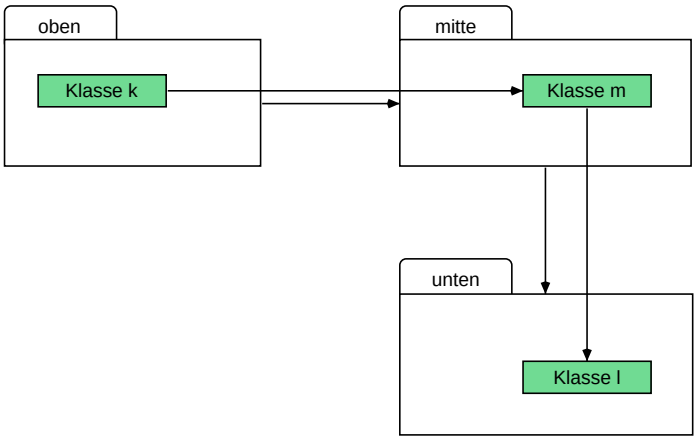


Abbildung 8.14: Die aus der Umordnung der Methodenaufrufe entstandenen Abhängigkeiten

`de.renew.remote` respektive `de.renew.gui`. Die Pakete `oben` und `mitte` aus der allgemeinen Erörterung fallen hier also zusammen.

Die Aufrufe, die in Abb. 8.12 dargestellt sind, haben ebenfalls eine Entsprechung in *Renew*. Bei der Methode, die `MultipleTokenFigure` auf `ObjectAccessor` aufruft, handelt es sich um `getFigure`; dabei wird der `RenewMode` als Argument übergeben. Auf diesem ruft dann der `ObjectAccessor` die Methode `getFigure` auf, die er dann als Ergebnis zurück liefert. Abbildung 8.15 stellt diesen Ablauf dar.

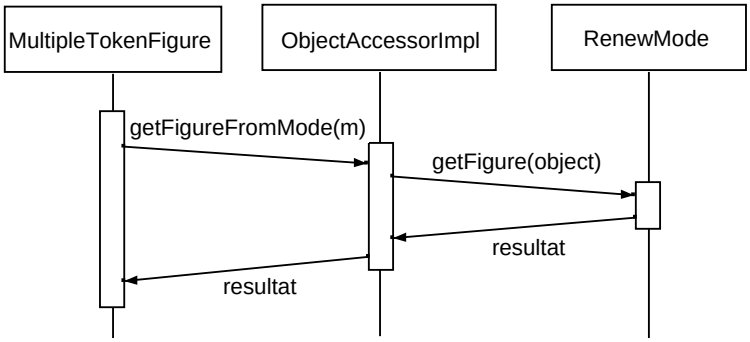


Abbildung 8.15: Die Aufrufreihenfolge zwischen `MultipleTokenFigure`, `ObjectAccessorImpl` und `RenewMode`

Für den Umbau erhält `RenewMode` eine Methode `getFigure` mit dem Parameter `ObjectAccessor`. Diese Methode wird nun von `MultipleTokenFigure` aufgerufen und delegiert diesen Aufruf an die ebenfalls neue Methode `getObject` aus dem `ObjectAccessorImpl`, der ihr als Argument übergeben wurde. Mit den daraus erhaltenen Daten kann die Funktion ihr Ergebnis berechnen und zurückliefern. Dieser neue Ablauf ist auf Abb. 8.16 zu sehen.

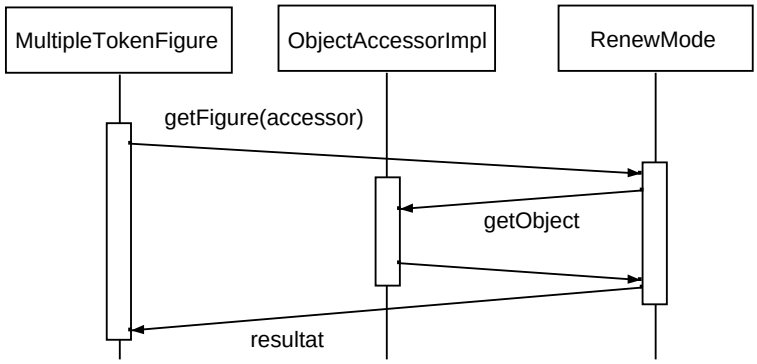


Abbildung 8.16: Sequenzdiagramm der Methodenaufrufe in *Renew* nach Umstellung gemäß Abschnitt 8.4.4

8.5 Zusammenfassung

Dieses Kapitel beschreibt die Aufteilung von *Renew* in Komponenten. Bei der Auswahl, welche dabei gebildet werden sollen, gibt es zwei Hauptkriterien.

Zum einen ist es möglich, Komponenten so zusammenzustellen, die zwischen *package*s bestehenden zyklischen Abhängigkeiten zusammenzufassen. Der Vorteil dieser Lösung ist, dass dabei zwischen den Komponenten keine zyklischen Abhängigkeiten entstehen; andererseits zeigt sich, dass die so entstehenden Komponenten nicht granular genug sind.

Zum Anderen bietet sich eine rein fachliche Aufteilung des Systems an, die sich prinzipiell beliebig fein vornehmen lässt, also bei objektorientierten Systemen bis auf Klassenebene; im Falle von *Renew* werden vorhandene Java-*package*s zusammengefasst. Um diese Aufteilung vornehmen zu können, ist außerdem Wissen über die Anwendungsdomäne des Systems notwendig, was bei dem ersten Ansatz nicht der Fall ist.

Anschließend werden Vorgehensweisen beschrieben, mit denen Abhängigkeitszyklen, die eventuell durch die fachliche Komponentisierung zwischen den Komponenten entstehen, entfernt werden können.

Als Ergebnis der praktischen Arbeit, die dieses Kapitel beschreibt, liegen Komponenten vor, die die Funktionalität von *Renew* enthalten. Diese Komponenten können nun, mit den entsprechenden Metainformationen versehen, von der Komponentenverwaltung aus dem vorigen Kapitel geladen werden.

Dadurch steht wieder der größte Teil der ursprünglich von *Renew* angebotenen Möglichkeiten zur Verfügugung. Einige Einschränkungen, die durch das Entfernen der Abhängigkeitszyklen entstanden sind, werden im nächsten Kapitel behoben. Dazu werden die Komponenten mit Erweiterungsschnittstellen versehen, so dass verloren gegangene Funktionalität durch Plugins wieder eingebracht werden kann.

Kapitel 9

Erweiterbarkeit der Komponenten

Wie im letzten Kapitel beschrieben, geht durch die Aufteilung in Komponenten einige Funktionalität des Systems verloren. Dies liegt daran, dass Verbindungen zwischen Klassen aus verschiedenen Komponenten entfernt werden, um zyklische Abhängigkeiten zu beseitigen. Um diese Funktionalität wieder herzustellen, wird nun der Plugin-Mechanismus verwendet, indem für die entsprechenden Komponenten Erweiterungsschnittstellen bereit gestellt werden.

Der Entwurf dieser Schnittstellen ist das Thema dieses Kapitels. Sie sollen es ermöglichen, die durch die Komponentisierung im letzten Kapitel verlorenen Funktionen durch Plugins wieder in die entsprechenden Komponenten einzufügen.

Als Bedingung für den Entwurf der Schnittstellen ist die von Szyperski [Szy02, S. 95] angegebene Definition der unabhängigen Erweiterbarkeit maßgeblich:

> A system is independently extensible if it is extensible and if independently developed extensions can be combined.

Es soll also möglich sein, Erweiterungen aus verschiedenen Quellen in das System zu bringen, ohne dass diese voneinander abhängen oder einander stören. Die dafür notwendige lose Kopplung wird durch den Einsatz von *Entwurfsmustern* [GHJV95] umgesetzt.

Als Ergebnis dieses Kapitels liegt *Renew* in der gewünschten Komponentenform vor, jedoch ohne die Einschränkungen in der Funktionalität, wie sie noch im vorigen Kapitel auftreten: die aus den Komponenten entfernten Funktionen lassen sich nun durch Plugins wieder einbringen.

Es ist jedoch wichtig zu bemerken, dass sich durch die dabei eingeführte allgemeine Erweiterbarkeit eben nicht nur bereits vorhandene, sondern auch neue Funktionalität hinzufügen lässt. Damit ist das Entwicklungsziel, *Renew* erweiterbar zu machen, erfüllt.

9.1 Verwendung von Entwurfsmustern

In diesem Abschnitt geht es um allgemeine Methoden, mit denen erweiterbare Software entworfen werden können. Besonders wichtig ist es, wegen der in Abschnitt 3.1 geforderten nichtzyklischen Abhängigkeitsrelation, dass die zu erweiternde Klasse die erweiternde nicht kennt, also lediglich über Interfaces auf sie zugreifen kann. Diese Art der losen Kopplung wird von den in [GHJV95] entwickelten *Entwurfsmustern* propagiert. Da sich diese auf objektorientierte Systeme beziehen, können sie auch auf die Repräsentantenobjekte der Komponenten angewendet werden.

Daher werden einige dieser Muster vorgestellt und auf ihre Anwendbarkeit im angestrebten komponentenorientierten *Renew* hin untersucht. Damit werden die grundlegenden Strukturen dargelegt, die für den Entwurf der späteren Klassen, die die Plugins gegenüber dem System vertreten, sinnvoll sind.

9.1.1 Chain of Responsibility

Die *Verantwortungskette* eignet sich dafür,

> eine Kopplung zwischen dem Sender einer Anfrage und ihrem Empfänger zu vermeiden, indem mehr als einem Objekt die Möglichkeit gegeben wird, auf die Anfrage zu reagieren (übersetzt von [GHJV95, Seite 223]).

Dies geschieht dadurch, dass eine Menge von Objekten vorhanden ist, die die Anfrage möglicherweise behandeln können. Diesen wird die Anfrage der Reihe nach vorgelegt, bis eines der Objekte sie bearbeiten kann.

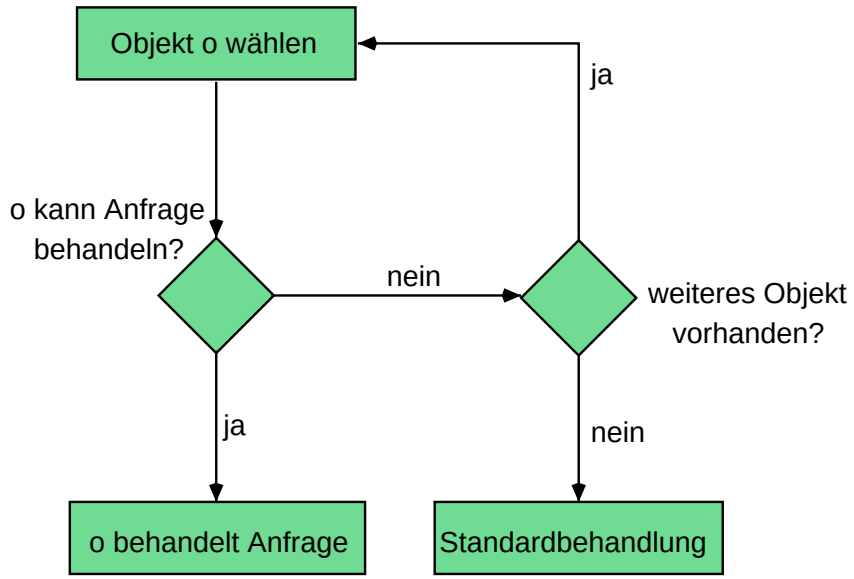


Abbildung 9.1: Verwendung der Chain of Responsibility

Verwendung für Komponenten

Wie kann dieses Muster nun verwendet werden, eine Komponente erweiterbar zu machen?

Anmelden der Empfänger In diesem Fall ist der Sender der Anfrage das Repräsentantenobjekt der Komponente und die möglichen Empfänger die Erweiterungen. Damit diese die Gelegenheit bekommen können, die Anfrage zu erhalten, müssen sie sich bei dem Sender bekannt machen. Dafür muss dieser über eine Methode in der Art `addReceiver` verfügen, die in der Schnittstelle eines ihrer Dienste vorhanden ist. Die darüber angemeldeten Objekte werden zur späteren Verwendung gespeichert.

Prüfen der Empfänger Irgendwann wird die erweiterte Komponente eine Anfrage generieren. Zu diesem Zeitpunkt geht sie die Liste der registrierten Objekte durch (vgl. Abb. 9.1). Damit sie entscheiden kann, ob ein Objekt die Anfrage behandeln kann, muss das Kettenmitglied eine entsprechende Abfrage ermöglichen.

Aufruf des korrekten Objektes oder Standardbehandlung Wenn sich ein Objekt aus den angemeldeten Erweiterungen bereit erklärt, die Anfrage zu behandeln, kann dieses verwendet werden. Anderenfalls sollte die Komponente eine Standardreaktion vorsehen, damit ein problemfreier Ablauf des Programms gewährleistet ist.

Nachteile der Chain of Responsibility

Dadurch, dass die Kopplung der Systembestandteile durch Entwurfsmuster gelockert wird, entstehen für das System mit der neuen Struktur auch Nachteile, die nicht unerwähnt bleiben dürfen.

Problem der Reihenfolge Ein Problem der Chain of Responsibility ist, dass keine Reihenfolge zwischen den Elementen etabliert werden kann, in der die Objekte die Anfrage erhalten. In [GHJV95] wird dieses Entwurfsmuster als eine Verkettung von hierarchisch angesiedelten Objekten vorgestellt. Diese Hierarchie besteht in dem hier vorliegenden Zusammenhang mit Plugins nicht, da es keine Ordnung der erweiternden Komponenten geben kann.

Dies liegt daran, dass die Erweiterungen völlig unabhängig voneinander entwickelt werden können und daher keine Priorität berechnet werden kann, welche der Komponenten zuerst zur Behandlung einer Anfrage aufgerufen werden soll.

Als Resultat kann es passieren, dass ein Objekt aus der Kette die Verantwortung für die Anfrage übernimmt, obwohl ein anderes besser geeignet wäre.

Laufzeitverhalten Ein weiterer Nachteil der Chain of Responsibility entsteht, wenn sich viele Plugins als Erweiterung bei einer Komponente anmelden. Im schlechtesten Fall müssen Objekte gefragt werden, ob sie eine Anfrage behandeln können. Insgesamt sind $n/2$ Aufrufe zu erwarten, bis das korrekte Objekt gefunden wird. Dadurch entstehen einerseits die Kosten der überflüssigen Methodenaufrufe, zum anderen wird auch von den einzelnen Objekten Rechenzeit gebraucht, um zu entscheiden, ob sie die Anfrage behandeln.

9.1.2 Command

Das *Kommando*-Entwurfsmuster funktioniert durch die "Kapselung einer Anfrage in einem Objekt". Dadurch "können Clients mit verschiedenen Anfragen [...] parametrisiert werden." (übersetzt aus [GHJV95, Seite 233]). Dies ist besonders für Klassen geeignet, die für eine Interaktion mit dem Benutzer zuständig sind. In JHotDraw etwa wird das *Command* verwendet, um auszuführenden Code zu Menüeinträgen zuzuordnen.

Verwendung für Komponenten

Auch in einem Pluginsystem sind *Commands* am sinnvollsten, wenn Komponenten mit dem Benutzer interagieren. Denkbar sind etwa Situationen, in denen eine Komponente eine Liste von auswählbaren Funktionen darstellt, die bei ihrer Wahl durch einen Aufruf eines Command-Objektes ausgeführt werden.

Anmelden des Kommandos Beim Initialisieren des erweiternden Plugins meldet dieses bei dem erweiterten ein Command-Objekt an. Dafür benötigt dieses eine Methode, die dies ermöglicht, etwa `addCommand`.

Auswahl eines Kommandos Damit der Benutzer ein Kommando auswählen kann, muss ihm eine Liste der verfügbaren Funktionen angeboten werden. Das bedeutet, dass zu der Kommando-Schnittstelle Methoden gehören sollten, die beispielsweise den Namen des Kommandos sowie eine menschenlesbare Beschreibung liefern.

Ausführen des gewählten Kommandos Wenn der Benutzer ein vorhandenes Kommando auswählt, muss die gewählte Funktionalität ausgeführt werden. Dazu benötigt das Kommando eine `execute`-Methode.

9.1.3 Observer

Das *Observer*-Muster eignet sich dafür, mehrere Objekte davon in Kenntnis zu setzen, dass ein bestimmtes Ereignis eingetreten ist. Javas Abstract Windowing Toolkit AWT verwendet eine Abwandlung des Observer-Patterns, den *Listener*.

Verwendung für Komponenten

Auch in einem Pluginsystem kommt es vor, dass eine Komponente ein Ereignis auslöst, auf das andere reagieren wollen.

Anmelden des Observer Analog zu den oben genannten Anwendungen von Mustern benötigt die zu erweiternde Komponente eine Methode `addObserver`.

Aufruf beim Eintritt des Ereignis Wenn ein Ereignis bei der erweiterten Komponente eintritt, für das Observer angemeldet sind, benachrichtigt sie diese der Reihe nach. Dafür steht bei den registrierten Objekten eine Methode zur Verfügung.

Im Unterschied zu der oben erwähnten Chain of Responsibility werden bei diesem Muster in jedem Falle alle der Observer benachrichtigt, während die Anfrage bei der Kette nur von einem der Objekte behandelt wird.

9.2 Erweiterbare Komponenten in *Renew*

Durch die Auftrennung *Renew*s in Komponenten ist ursprünglich vorhandene Funktionalität verloren gegangen, weil die Verbindung zwischen Klassen getrennt wurde. Diese kann mit den im vorigen Abschnitt vorgestellten Mitteln wieder hergestellt werden. Dazu werden die Entwurfsmuster auf die Repräsentanten der Komponenten angewendet.

In Abschnitt 8.2 wurde besonders für zwei Komponenten gefordert, dass sie eine eigene Komponente bilden sollen. Dabei handelte es sich einerseits um die grafische Benutzungsoberfläche, andererseits um den Compiler, der zum Übersetzen der Netze in eine simulierbare Form verwendet wird.

Dieser Abschnitt beschreibt, wie die Schnittstellen für die entsprechenden Komponenten mit Hilfe der im vorigen Abschnitt beschriebenen Methoden so entworfen werden können, dass die ursprüngliche Funktionen wieder verfügbar wird.

9.2.1 Gui

Die grafische Benutzungsoberfläche lässt sich an vielen Stellen erweitern. Zwei davon sind bereits in vorigen Kapiteln zur Sprache gekommen und sollen deswegen hier behandelt werden.

Menüs und Paletten

Schon im Prototyp gilt die Anforderung, die grafische Oberfläche mit zusätzlichen Elementen zu versehen, die dem Benutzer eine Interaktion ermöglicht. Dabei ist in der Vorversion die Plugin-Verwaltung dafür zuständig, die Einträge neuer Menüs und Paletten in die Gui vorzunehmen.

In der komponentisierten Fassung von *Renew* kann dies nicht mehr geschehen, da die Verwaltung lediglich zum Laden und Verwalten der Plugins zuständig ist und die Gui nicht notwendigerweise zur Verfügung steht. Daher muss schon beim Entwurf der **Gui**-Komponente dafür gesorgt werden, dass sie über die entsprechenden Methoden verfügt.

Entwurfsmuster Für die Einführung von Menüeinträgen ist das *Command*-Muster geeignet. Dabei kann die erweiternde Komponente ein Command erzeugen, das bei Auswahl des neuen Menü-Eintrags durch den Benutzer ausgeführt werden kann. Dies würde die Einführung einer Methode mit der Signatur

```
addMenu(String label, Command toExecute)
```

in der Schnittstelle der **Gui**-Komponente nahe legen; damit ist diese für die Erzeugung des Menüs selbst zuständig. In der Tat wird dieses Musters im JHotDraw-Framework genau so benutzt; diese ist jedoch auf die Verwendung durch Unterklassen eingeschränkt.

Das Problem an diesem Ansatz ist, dass so das erweiternde Plugin keinen Zugriff auf das erzeugte Menü hat und dieses daher nicht verändern kann, etwa um Untermenüs einzutragen und zu entfernen; dies schränkt die Flexibilität dieser Lösung ein.

Als Alternative kann das Plugin das Menü erzeugen und dann der Gui übergeben, die es dann in das vorhandene Menü einfügt. Dies ist die implementierte Variante.

Umsetzung Die konkrete Umsetzung in dem Repräsentantenobjekt für die Gui, `de.renew.gui.GuiPlugin`, besteht aus Methoden, mit denen die Menü- bzw. Palettenobjekte registriert sowie wieder entfernt werden können:

- `addMenu(java.awt.Menu m)`
- `removeMenu(java.awt.Menu m)`

Darstellung von Simulationselementen

In Abschnitt 8.3.4 wird die Trennung der **Gui**- von der **Formalism**-Komponente dadurch erzwungen, dass auf die von den *Mode*s zur Verfügung gestellten speziellen Darstellungsmöglichkeiten verzichtet wird.

An dieser Stelle soll wieder die ursprüngliche Funktionalität hergestellt werden, bei der ein Formalismus für bestimmte Tokens eine eigene Darstellung erzeugen kann. Auch diese Darstellungsmöglichkeiten müssen der **Gui**-Komponente über ihre Schnittstelle mitgeteilt werden.

Die Struktur der Verantwortlichkeiten ist in Abbildung 9.2 zu sehen: Formalismus-Komponenten definieren die Compiler, die sie dem **Formalism**-Plugin zur Verfügung stellen. Optional können sie der **Gui**-Komponente ein Objekt übergeben, das die Darstellung von speziellen Simulationselementen für den jeweiligen Formalismus ermöglicht.

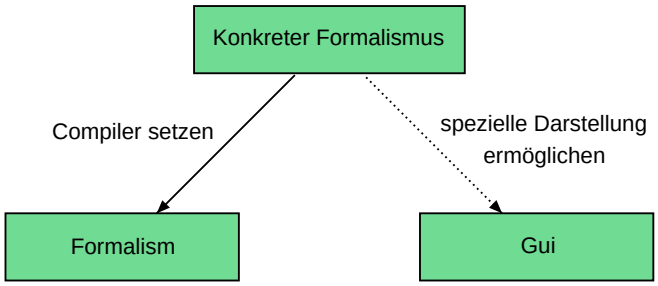


Abbildung 9.2: Aufgaben einer Formalismus-Komponente: Setzen eines Compilers, optional spezielle Darstellung anmelden

Entwurfsmuster Als konkrete Umsetzung der Darstellung ist das Muster *Chain of Responsibility* verwendbar. Dabei ist die Anfrage, die an die verschiedenen Kettengliedern gestellt wird, die Aufforderung, ein Element zu zeichnen.

Umsetzung Um eine Verantwortungskette aufzubauen, muss für die Objekte, die darin enthalten sein können, ein einheitliches Interface entworfen werden.

Dieses muss zum einen eine Abfrageoperation haben, mit der festgestellt werden kann, ob das entsprechende Objekt die Darstellung übernimmt. Da nicht festgelegt ist, welche Art von Token während der Simulation dargestellt werden müssen, ist hier als Parameter die Oberklasse aller Java-Klassen zu übergeben, `java.lang.Object`.

Die Methoden des *Mode*s, der durch diese Gui-Erweiterung ersetzt wird, produzieren ein Objekt der Klasse `CH.ifa.draw.framework.Figure`; dies ist in JHotDraw das grundlegende Interface für darstellbare Elemente. Um möglichst wenige Veränderungen vorzunehmen, kann auch das angestrebte Interface eine `Figure` erzeugen. Die

Gui-Komponente verwendet dann die Figure als Darstellung für das Token. Hier ist als Parameter das gleiche Objekt zu übergeben, das für die vorherige Abfrage verwendet wurde.

Insgesamt besteht also das Interface aus den Methoden:

- `boolean canDisplay(Object o)`
- `CH.ifa.draw.framework createDisplay(Object o)`

Als Standarddarstellung verwendet die Gui die Textdarstellung des anzuzeigenden Objekte, die aus dessen `toString()`-Methode erzeugt wird.

9.2.2 Formalismen

Abschnitt 8.2 beschreibt die Forderung, dass es im umgebauten System möglich sein soll, dass der Benutzer die zur Verfügung stehenden Compiler zur Laufzeit auswählen kann.

Anforderungen

Was ist zu beachten, um dies zu ermöglichen? Dabei ist es sinnvoll, die vorherige Struktur zu betrachten. Ein Klassendiagramm mit den an der Simulation eines Netzsystems beteiligten Klassen ist auf Abb. 9.3 zu sehen.

Wie weiter oben beschrieben, ist die auf diesem Bild gezeigte Verwendungsbeziehung zwischen `CPNApplication` und `RenewMode` nach der Aufteilung von *Renew* nicht mehr vorhanden. Vielmehr befindet sich der Compiler, den der *Mode* zur Verfügung stellt, nun in der **Formalism**-Komponente.

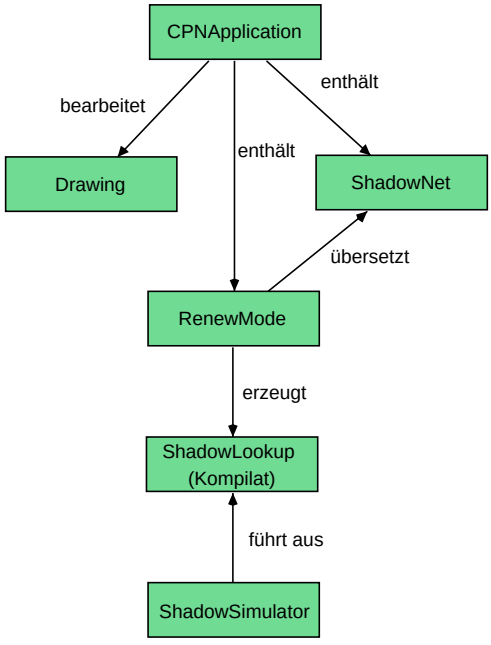


Abbildung 9.3: Beziehungen zwischen den an der Simulation eines Netzes beteiligten Klassen

Die neuen Beziehungen sind auf Abb. 9.4 gezeigt. Hier ist die **Formalism**-Komponente unabhängig von `Gui`, **NeuerFormalism** repräsentiert ein Plugin, die einen Compiler zur Verfügung stellt. Das **FormalismGui**-Plugin, die in einem helleren Rechteck dargestellt ist, fungiert als Vermittler: Es versorgt die Gui mit einem Menü, in dem die vorhandenen Formalismen angezeigt werden; wenn der Benutzer einen auswählt, wird in der **Formalism**-Komponente der entsprechende Compiler gesetzt.

Entwurfsmuster

Bei den Formalismen kann es von Bedeutung sein, dass eine Komponente davon erfährt, wenn ein bestimmter Compiler hinzugefügt, entfernt oder ausgewählt wurde. Dies wäre beispielsweise bei **FormalismGui** in Abb. 9.4 der Fall, damit diese das angezeigte Menü aktuell halten kann.

Dies ist ein typischer Anwendungsfall für das Muster *Observer*. Hier sind die Clients daran interessiert, über eine Zustandsveränderung in dem Formalism-Plugin benachrichtigt zu werden.

Umsetzung

Zum einen wird in der Schnittstelle des Formalism-Plugins eine Möglichkeit benötigt, neue Compiler zur Verfügung zu stellen. Die dazu verwendeten Methoden sind, analog zu den vorigen Fällen:

- `addCompiler(ShadowCompiler compiler)` und
- `removeCompiler(ShadowCompiler compiler)`

Zusätzlich muss die oben geforderte Unterstützung des Observer-Patterns umgesetzt werden; auch für diese gibt es Methoden zum Hinzufügen und Entfernen der Observer. Als Typ für Objekte, die als Observer zugelassen werden, dient das Interface `de.renew.formalism.FormalismChangeListener` mit der Methode `formalismChanged()`.

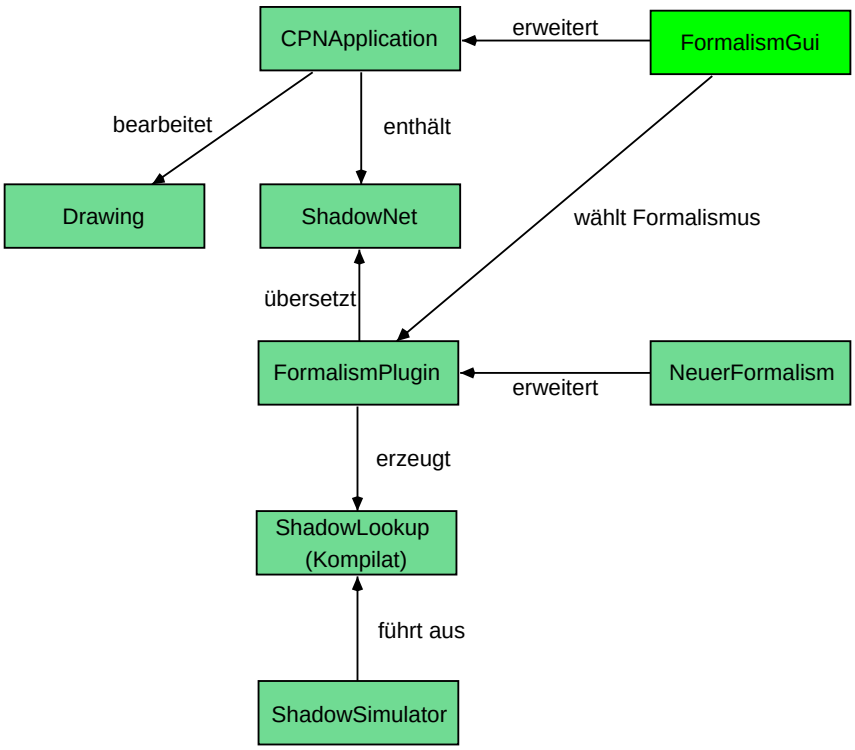


Abbildung 9.4: Beziehungen zwischen den an der Simulation eines Netzes beteiligten Klassen nach der Komponentisierung

9.3 Bewertung

Erfüllt das für *Renew* umgesetzte Pluginsystem die eingangs aufgestellten Anforderungen *Verwaltbarkeit*, *Konfigurierbarkeit* und *Erweiterbarkeit*?

Die Umstellung von *Renew* in fachliche Komponenten setzt *Verwaltbarkeit* um, indem eine weitere Hierarchieebene zur Verwaltung eingeführt wird. Auch die *Konfigurierbarkeit* des Systems wird erhöht, da sich der Benutzer aussuchen kann, welche Komponenten er benötigt. Reine Komponenten reichen dann nicht aus, wenn die einzelnen Komponenten erweiterbar gemacht werden sollen, genauer: wenn Komponenten von anderen um Funktionalität ergänzt werden können.

In diesem Fall eignen sich Plugins sehr gut. Dafür müssen die erweiterbaren Komponenten über von ihnen bereit gestellte Dienste Plugins aufnehmen können. Dadurch ist es möglich, dass zwei Komponenten miteinander kommunizieren, ohne über die Verwaltung zu gehen; dies ist effizienter und für den Entwickler einfacher einzusetzen.

Das in Kapitel 5 vorgestellte Modell für Komponenten und Plugins hat sich als guter Leitfaden für die spätere Entwicklung des Pluginsystem für *Renew* erwiesen.

Zum praktischen Teil ist zu sagen, dass sich die Einteilung der in *Renew* vorhandenen Klassen und *package*s in Komponenten als sehr aufwändig herausgestellt hat. Besonders das Entfernen der zyklischen Abhängigkeiten ist nur für ein bereits vorher gut strukturiertes System sinnvoll durchführbar. Außerdem ist es unerlässlich, bereits fachliches Wissen über die Domäne der Software zu besitzen, die komponentisiert werden soll; nur so kann sicher gestellt werden, dass die entstehenden Komponenten fachliche Einheiten bilden und dass für sie angemessene Erweiterungsschnittstellen entworfen werden können.

Für *Renew* besonders hervorzuheben ist die Anforderung, dass der Benutzer den *Mode* zur Laufzeit des Systems ändern können soll; das im Zuge dieser Arbeit entstandene Pluginsystem bietet diese Möglichkeit. Die ursprüngliche Struktur sieht eine Konfigurationsvariable vor, die jedoch nur zum Start des Programms ausgewertet wird. Insofern ist ein dynamischer Wechsel des *Mode*s grundsätzlich schon vorbereitet.

Die Komponentenbildung erleichtert die Trennung des *Mode*s in *GUI*- und *Formalismus*-Teil. Nach der Umsetzung dieser beiden Komponenten und dem Entwurf ihrer generischen Erweiterungsschnittstellen ist es überraschen einfach möglich, die verschiedenen *Mode*s, die nach der Komponententisierung zunächst nicht zur Verfügung stehen, wieder einzubinden. Daher lässt sich annehmen, dass sich auch zukünftig zu entwickelnde Netzformalismen problemlos in den entstandenen Plugin-Mechanismus einbinden lassen werden.

Zusammenfassend lässt sich also sagen, dass durch den Umbau von *Renew* in eine Komponentenarchitektur die vorher gestellten Anforderungen der *Verwaltbarkeit*, *Konfigurierbarkeit* und der *Erweiterbarkeit* erfüllt werden. Dabei wird besonders die Ersetzung der *Mode*s durch den flexibleren Mechanismus, Netzformalismen zu verwenden, für die weitere Entwicklung von *Renew* sicher eine wichtige Rolle spielen.

9.3.1 Vergleich mit *eclipse*

Das entwickelte System bietet die Möglichkeit, Plugins dynamisch zu laden und zu entladen. Dabei ist nicht fest gelegt, welche Funktionen die geladenen Komponenten zur Verfügung stellen. Dies bedeutet, dass die Anwendung des vorhandenen Mechanismus nicht auf die Verwendung mit *Renew* beschränkt ist.

Dadurch drängt sich der Vergleich des Systems mit dem in der Arbeit vorgestellten *eclipse* auf. Dieses bietet Möglichkeiten zur Verwaltung der Plugins, die bei einer Arbeit

wie der vorliegenden naturgemäß nicht umzusetzen ist.

Andererseits gibt *eclipse* den Plugins ein starkes "Weltbild" vor, so dass ihre Flexibilität eingeschränkt ist. Dies liegt einerseits daran, dass ein umfassendes Rahmenwerk zur Verfügung steht, dessen Funktionalität die Komponenten nutzen können und sollen. Andererseits ermöglicht gerade diese Vorgabe eine deutlich genauere Steuerung der Kommunikation zwischen den Komponenten, als dies mit dem hier vorgestellten System der Fall ist.

9.3.2 Vergleich mit Agenten

In dem in Kapitel 5 erstellten Modell wird die Komponentenverwaltung als Plattform bezeichnet, die die Komponenten enthält und verwaltet. Diese Sicht dient dazu, das System mit in der Agentenorientierung entwickelten vergleichen zu können. Dabei wird das von der *Foundation for Intelligent Physical Agents, FIPA* [Fou03] entworfene Agentenkonzept herangezogen, das in MULAN [Röl99] umgesetzt wird. Welche Punkte sind bei einem solchen Vergleich zu berücksichtigen?

Agenten kommunizieren indirekt miteinander. Allgemein wird immer, wenn eine Nachricht versendet wird, ein Dienst namens *Message Transport Service, MTS* verwendet. Im hier entwickelten Pluginsystem gibt es einen solchen Dienst nicht; auf konzeptioneller Ebene entspricht er der Java Virtual Machine, die die Methodenaufrufe zwischen den Schnittstellen realisiert.

Um einen Dienst zu nutzen, fragen Agenten einen vordefinierten Dienst (den *directory facilitator, DF*) nach Agenten, die diesen anbieten. Dies entspricht im vorliegenden System der Anfrage, die eine Komponente an den `PluginManager` stellen muss, um Zugriff auf eine Komponente mit einer gewissen Funktionalität zu erhalten.

Im Falle von Agenten können diese, wenn sie eine positive Antwort erhalten haben (also einen anderen Agenten, der den gewünschten Dienst anbietet), die Kommunikation beginnen. Dabei wird die Kommunikation nicht mehr über den DF ausgeführt, sondern direkt zwischen den Agenten. Auch in diesem Punkt stimmen die Abläufe mit dem *Renew*-Pluginsystem überein.

Andererseits werden Plugins im erstellten System von einer zentralen Stelle verwaltet und können mehrere Komponenten erweitern. Dies verhindert, dass eine Komponente selbst als Plattform gesehen wird, da sich in diesem Fall ein Plugin nur in einer Komponente aufhalten könnte.

Es kann also gesagt werden, dass in dem in MULAN propagierten Agentenmodell und dem implementierten Pluginsystem ähnliche Konzepte verwendet werden; sie stimmen jedoch nicht vollständig überein.

Kapitel 10

Fazit

Dieses Kapitel bildet den Abschluss der vorliegenden Arbeit. Es enthält eine Zusammenfassung, in der die Ergebnisse der vorherigen Kapitel noch einmal kurz dargestellt werden.

Anschließend werden im Ausblick einige weitere Ideen vorgestellt, die durch die Umsetzung des Plugin-Mechanismus in *Renew* ermöglicht werden, im Rahmen dieser Arbeit jedoch aus thematischen oder zeitlichen Gründen nicht angegangen wurden.

10.1 Zusammenfassung

Diese Arbeit beschreibt, wie der Aufbau des Petrinetzsimulators *Renew* in eine Plugin-basierte Architektur umgewandelt wird. Dabei wird die Anforderung umgesetzt, Konfigurierbarkeit, Verwaltbarkeit und Erweiterbarkeit der Software zu verbessern.

Die Arbeit teilt sich in einen konzeptionellen und einen praktischen Teil. Im ersten Teil werden Grundlagen zum Konzept der komponentenorientierten Softwareentwicklung vorgestellt. Dabei werden Komponenten als Softwarekonstrukte definiert, die bestimmte Dienste zur Verfügung stellen und deren Kommunikation über eine gemeinsame Stelle geregelt wird.

Die Erörterung, inwiefern die Komponentenorientierung die vorher angegebenen Anforderungen erfüllen kann, führt zu der Feststellung, dass Komponenten geeignet sind, Konfigurierbarkeit und Verwaltbarkeit umzusetzen; für die Erweiterbarkeit jedoch sind weitere Ansätze nötig.

Diese finden sich in der Plugin-Architektur. Bei Plugins handelt es sich um Komponenten, die Schnittstellen anbieten, mit denen ihre Funktionalität erweitert werden kann; zur Veranschaulichung dieses Prinzips werden zwei existierende Pluginsysteme vorgestellt. Den Abschluss des zweiten Teils bildet als Ergebnis das Modell eines erweiterbaren Systems.

Im zweiten Teil werden die vorher entwickelten Konzepte in *Renew* implementiert. Dafür wird zunächst das Pluginsystem entwickelt, das die *Renew*-Komponenten verwaltet. Dabei bieten Komponenten ihre Dienste über Schnittstellenobjekte an, die von der Plattform verwaltet werden. Dienste, durch Strings identifiziert, werden in der Metainformationsdatei der Komponente aufgelistet; hier werden ebenfalls eventuelle Abhängigkeiten der Komponente deklariert. Die Verwaltung des Lebenszyklus einer Komponente geschieht über das Java-Interface `IPlugin`, das die Methoden `init()` und `shutdown()` enthält. Die Plugin-Verwaltungsinstanz lädt eine Komponente in drei Schritten: finden, sortieren und laden. Für jede dieser Aufgaben ist eine eigene Klasse zuständig; die Exemplare dieser Klassen werden vom `PluginManager`, der Verwaltungsinstanz, gesteuert.

Das umgesetzte Verwaltungssystem ist sehr flexibel, da es in der Lage ist, allgemeine Komponenten – d.h. solche mit beliebiger Funktionalität – zu laden. Auf der anderen Seite ist es so implementiert, dass Laufzeitoverhead und zusätzlicher Speicherbedarf minimal sind.

Der zweite große Teil der praktischen Arbeit besteht darin, die in *Renew* vorhandenen Bestandteile in Komponenten aufzuteilen. Da sich die gewählte Komponentisierung hauptsächlich an fachlichen Aspekten orientiert, müssen in diesem Zusammenhang abschließend zyklische Abhängigkeiten aus den identifizierten Komponenten beseitigt werden. Durch die dabei entstehenden Komponenten werden die Anforderungen der Verwaltbarkeit und Konfigurierbarkeit umgesetzt.

Da das System in Komponenten aufgeteilt vorliegt, werden für diese nun Schnittstellen definiert. Dabei wird darauf geachtet, dass diese gemäß der Anforderungen an ein Plugin-System Möglichkeiten zur Erweiterung der Komponente enthalten, um die letzte Anforderung zu erfüllen.

Als Ergebnis liegen Komponenten vor, in denen die ursprüngliche *Renew*-Funktionalität uneingeschränkt zur Verfügung steht, wenn sie mit der vorher erstellten Plugin-Verwaltung eingesetzt werden. Tatsächlich zeigt sich, dass die Benutzung von *Renew* als Pluginsystem zwar die erwarteten Vorteile der zusätzlichen Flexibilität und Konfigurierbarkeit bietet, sich jedoch nicht negativ auf die Ausführungsgeschwindigkeit auswirkt.

Damit sind die gestellten Anforderungen, *Renew* verwaltbarer, konfigurierbarer und einfacher erweiterbar zu machen, erfüllt.

10.2 Ausblick

Punkte wie einfachere Verwaltbarkeit des Sourcecodes und einfachere Erweiterbarkeit des Gesamtsystems sind Vorteile, die die Entwicklung von *Renew* allgemein unterstützen. Es lassen sich jedoch auch wünschenswerte spezielle Anforderungen formulieren, die durch die Verwendung von Komponenten einerseits, die Ausrichtung am Plattform-Paradigma andererseit erst möglich werden.

10.2.1 Weitere Komponentisierung

Die Komponentisierung, wie sie in Kapitel 8 vorgestellt wurde, ließe sich noch weiter verfeinern. Dazu wären weitere Umbauarbeiten notwendig, die sich während der Durchführung dieser Arbeit nicht mehr umsetzen ließen.

Abspaltung einer I/O-Komponente Das Speichern und Laden von Netzen ist zu großen Teilen in die grafische Oberfläche integriert. In diesem Fall wäre eine eigenständige Komponente sinnvoll, die für die Ein- und Ausgabe generell zuständig ist.

Ein Formalismus pro Netz Der Simulatorkern kann Netze ausführen, ohne dass dies vom verwendeten Compiler abhängig ist. Dies macht es denkbar, dass, wenn mehrere Netze erstellt werden, für jedes ein eigener Formalismus eingesetzt werden kann; im gegenwärtigen Zustand ist lediglich die Verwendung eines Formalismus für alle geladenen Netze vorgesehen. Für die Umsetzung dieses Vorschlages ist es jedoch nicht ausreichend, ein neues Plugin zu entwickeln. Vielmehr muss dies in den bereits vorhandenen Komponenten, die am Simulationsprozess beteiligt sind, umgesetzt werden.

10.2.2 Weitere Plugins

Während die Abspaltung weiterer Komponenten aus dem bestehenden System eher der Verwaltbarkeit des Codes dient, bietet das in *Renew* eingeführte Pluginsystem auch die Möglichkeit, neue Funktionalität einzubringen, indem neue Komponenten oder Plugins für bestehende entworfen werden.

Entwicklung einer Konfigurationskomponente Die Konfiguration der einzelnen Komponenten innerhalb von *Renew* wird über die Metainformationen der entsprechenden Plugins geregelt. Eine Komponente, die die bestehenden Plugins und ihre Einstellungen verwaltet, würde den Benutzern den Umgang mit dem System erleichtern.

Einbau eines Logging-Konzeptes Während der Simulation eines Netzsystems ist die von *Renew* erzeugte Ausgabe so ausführlich, dass sie für ein ernsthaftes Debugging nicht zu gebrauchen ist. Ein Plugin, das die Ausgaben filtert und dadurch benutzbarer macht, könnte dieses Problem beheben.

Einbindung der *Renew*-Komponenten in *eclipse* Da bei der Aufteilung von *Renew* nicht auf den vorher entworfenen, spezifischen Plugin-Mechanismus hingearbeitet wurde, sollte es verhältnismäßig einfach möglich sein, die entstandenen Komponenten für eine andere Verwaltung vorzubereiten. Dies ist besonders bei *eclipse* interessant, da dieses System bereits über viele wünschenswerte Funktionen einer Entwicklungsumgebung verfügt, die dann für die Verwendung von Mulan nicht neu programmiert werden müssten.

10.2.3 Agenten als Komponenten

Dadurch, dass die Plugin-Architektur strukturell an das Agentenframework MULAN angelehnt ist, ist es denkbar, dass eine Komponente entwickelt wird, die eine Agentenplattform steuert. Die darauf vorhandenen Agenten könnten dann ebenfalls in das Pluginsystem als Dienstanbieter eingebracht werden.

Für die Umsetzung dieser Idee sind große Umstrukturierungsarbeiten an *Renew* notwendig. Eine Komponente, in der eine MULAN-Agentenplattform läuft, benötigt eine laufende Simulation, da die MULAN-Plattform als ausführbare Referenznetze vorliegen. Das System kann im gegenwärtigen Zustand aber lediglich eine Simulation zur Zeit ausführen, d.h. eine laufende Agentenplattform würde den Start einer weiteren Simulation verhindern. Daher wäre es notwendig, den Simulator neu zu konzipieren, so dass er in der gleichen Virtual Machine mehrfach gestartet werden kann.

Konzeptionell läuft dieser Vorschlag darauf hinaus, das in Kapitel 5 vorgestellte Modell eines Plugin-Mechanismus durchgängig auf alle Ebenen der Softwareumgebung umzusetzen. Da es sehr allgemein anwendbar ist, kann es auch als Richtlinie dafür dienen, mit MULAN entworfene Agentensysteme erweiterbar zu machen; damit würde das vorgestellte Prinzip auf einer höheren Ebene umgesetzt werden.

Ideal wäre es, wenn auch die Ebenen, die unterhalb von *Renew* liegen, also etwa die *Java Virtual Machine*, ein eventuell vorhandenes *Fenstersystem*, oder – auf noch tieferer Ebene – das *Betriebssystem*, mit dem gleichen Verwaltungsmechanismus auf einheitliche Art und Weise erweiterbar wären. Das in Anschnitt 3.4 beschriebene *.NET* repräsentiert einen Ansatz in diese Richtung.

Bei Betrachtung der *.NET*-Architektur wird klar, dass die Umsetzung einer solchen Idee erheblichen Aufwand bedeutet und den Rahmen einer Diplomarbeit um Größenordnungen sprengt. Die Tatsache, dass die Firma Microsoft ein entsprechendes Projekt durchführt, ist jedoch als Indiz für die Bedeutung des Konzeptes zu deuten.

Literaturverzeichnis

[Apa03] Apache. Xerces2 Java Parser Webpage. `http://xml.apache.org/xerces2-j/index.html`, Apache Website, August 2003.

[Arg03] ArgoUML Project. Argo/UML Homepage. `http://www.argouml.tigris.org`, 2003.

[Ber91] H. Ronald Berlack. *Software Configuration Management*. John Wiley & Sons, September 1991.

[Bol03] Azad Bolour. Notes on the Eclipse Architecture. `http://www.eclipse.org/articles/Article-Plug-in-architecture/plugin_architecture.html`, Bolour Computing, 2003.

[Bor03] Borland. Borland Together Main Page. `http://www.borland.com/together/index.html`, 2003.

[Cab02] Lawrence Cabac. Entwicklung von geometrisch unterscheidbaren Komponenten zur Vereinheitlichung von Mulan-Protokollen. Studienarbeit, Universität Hamburg, 2002.

[Car01] Timo Carl. Evaluation und beispielhafte Erweiterung einer Referenznetzbasierten Agentenumgebung (Arbeitstitel). Studienarbeit, Universität Hamburg, Februar 2001.

[Car04] Timo Carl. Entwicklung eines verteilten und agentenbasierten Workflowsystems mit *Renew*. Diplomarbeit, Universität Hamburg, Februar 2004.

[CM 03] CM Magazin. CM Magazin Homepage. `http://www.cmmagazin.de`, CM Magazin, Juli 2003.

[Dar00] Susan Dart. *Configuration Management - the missing link in web engineering*. Artech House, November 2000.

[Dis03] Distributed Systems Research Group. SOFA Component Model Documentation Homepage. `http://nenya.ms.mff.cuni.cz/projects/sofa/tools/doc/compmodel.html`, Charles Universität Prag, September 2003.

[ecl03] eclipse Projekt. eclipse Homepage. `http://www.eclipse.org/`, *eclipse*, 2003.

[Eic02] Clemens Eichler. Entwicklung einer Plugin-Architektur für dynamische Komponenten. Diplomarbeit, vsis, Universität Hamburg, Februar 2002.

[equ03] equinox-Projekt. equinox Homepage. `http://www.eclipse.org/equinox`, *eclipse*, 2003.

[Fou03] Foundation for Intelligent Physical Agents. FIPA homepage. `http://www.fipa.org`, Foundation for Intelligent Physical Agents, 2003.

[Fow99] Martin Fowler. *Refactoring – Improving the Design of Existing Code*. Addison-Wesley Publishing Company, 1999.

[Fow03] Martin Fowler. Refactoring Online Catalog. `http://www.refactoring.com/catalog/`, ThoughtWorks, 2003.

[gA03] gentleware AG. Gentleware AG homepage. `http://www.gentleware.de`, Universität Hamburg, Fachbereich Informatik, 2003.

[GE04] Erich Gamma and Thomas Eggenschwiler. JHotDraw homepage. `http://www.jhotdraw.org`, 2004.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley Publishing Company, 1995.

[Gri98] Frank Griffel. *Componentware*. dpunkt Verlag, 1998.

[Gri01] Frank Griffel. *Verteilte Anwendungssysteme als Komposition klassifizierter Softwarebausteine*. Dissertation, Universität Hamburg, Fachbereich Informatik, 2001.

[IBM03a] IBM. XML Parser for Java Webpage. `http://www.alphaworks.ibm.com/tech/xml4j`, IBM AlphaWorks Website, August 2003.

[IBM03b] IBM Rational Software. IBM Rational Software homepage. `http://www.rational.com/products/rose/`, 2003.

[Jac02] Thomas Jacob. Implementierung einer sicheren und rollenbasierten Workflowmanagement-Komponente für ein Petrinetzwerkzeug. Diplomarbeit, Universität Hamburg, Fachbereich Informatik, 2002.

[Jen00] Nick R. Jennings. On agent-based software engineering. *Artificial Intelligence*, 117(2):277–296, 2000.

[Kal01] Tomas Kalibera. SOFA Support in C++ Environments. `http://sofa.forge.objectweb.org/index.phtml`, Universität Prag, 2001.

[Kum02] Olaf Kummer. *Referenznetze*. Logos Verlag, Berlin, 2002.

[KW03] Olaf Kummer and Frank Wienberg. Renew (Reference net workshop) homepage. `http://www.renew.de`, Universität Hamburg, Fachbereich Informatik, 2003.

[KWD03] Olaf Kummer, Frank Wienberg, and Michael Duvigneau. Renew - Architecture Guide. Dokumentation, Universität Hamburg, Fachbereich Informatik, 2003.

[LTQL02] Thuan L. Thai and Hoang Q. Lam. *.NET Framework Essentials*. O'Reilly, 2. edition, 2002.

[Mic03] Microsoft Cooperation. .NET Framework Homepage. `http://msdn.microsoft.com/netframework/`, Microsoft Developers' Network, September 2003.

[MP00] D. Mennie and B. Pagurek. An Architecture to Support Dynamic Composition of Service Components, 2000.

[MV03] Klaus Marquardt and Markus Völter. Plugins: Anwendungsspezifische Komponenten. *JavaSpektrum*, Februar 2003.

[No 03] No Magic, Inc. MagicDraw Homepage. `http://www.magicdraw.com`, No Magic, Inc., 2003.

[Obj03a] Object Management Group, Inc. OMG homepage. `http://www.omg.org`, Object Management Group, Inc., 2003.

[Obj03b] Object Management Group, Inc. OMG Unified Modelling Language Specification. Spezifikation, Object Management Group, Inc., März 2003.

[Obj03c] Object Management Group, Inc. UML homepage. `http://www.omg.org/uml`, Object Management Group, Inc., 2003.

[Oes98] Bernd Oestereich. *Objektorientierte Softwareentwicklung – Analyse und Design mit der Unified Modeling Language*. Oldenburg Verlag, 4. edition, 1998.

[PV02] Frantisek Plasil and Stanislav Visnovsky. Behavior protocols for software components. *IEEE Transactions on Software Engineering*, 28(11), Nov 2002.

[Ree04] Christine Reese. Multiagentensysteme: Anbindung der petrinetzbasierten Plattform *CAPA* an das internationale Netzwerk *Agentcities*. Diplomarbeit, Universität Hamburg, Februar 2004.

[Röl99] Heiko Rölke. Modellierung und Implementation eines Multi-Agenten-Systems auf der Basis von Referenznetzen. Diplomarbeit, Universität Hamburg, 1999.

[Sam96] Johannes Sametinger. *Software Engineering with Reusable Components*. Springer, 1996.

[sPP03] *SOFA* Prototyp Projekt. *SOFA* Java Prototyp. `http://sofa.forge.objectweb.org/index.phtml`, SourceForge, 2003.

[Szy02] Clemens Szyperski. *Component software: beyond object-oriented programming*. ACM Press books. Addison-Wesley, 2. edition, 2002.

[Tur96] Volker Turau. *Algorithmische Graphentheorie*. Addison-Wesley, 1996.

[VG03] Rüdiger Valk and Claude Girault. *Petri nets for systems engineering : a guide to modeling, verification and applications*. Springer, 2003.

[Wie01] Frank Wienberg. *Informations- und prozeßorientierte Modellierung verteilter Systeme auf der Basis von Feature-Structure-Netzen*. Dissertation, Universität Hamburg, Fachbereich Informatik, 2001.

Abbildungsverzeichnis

Danksagungen

Ich bedanke mich bei Dr. Daniel Moldt und Prof. Dr. Lamersdorff, dass sie die Betreuung meiner Diplomarbeit übernommen haben.

Weiterhin gilt mein Dank den Korrekturlesern dieser Arbeit, besonders Dr. Olaf Kummer und Michael Duvigneau für ihre wichtigen fachlichen Hinweise und Anregungen, aber auch Anja Hennemuth, Lawrence Cabac, Jan Ortmann sowie Norbert Schumacher.

Erklärung

Ich versichere, dass ich die vorstehende Arbeit selbstständig und ohne fremde Hilfe angefertigt und mich anderer als der im beigefügten Verzeichnis angegebenen Hilfsmittel nicht bedient habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht.

Ich bin mit der Einstellung der Arbeit in den Bestand der Bibliothek des Fachbereiches einverstanden.

Hamburg, den 3. Oktober 2003

Jörn Schumacher