



Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

ABSCHLUSSARBEIT

MODULARISIERUNG DES RENEW
PLUGIN SYSTEMS

ARKADI DASCHKEWITSCH

12. SEPTEMBER 2019 UNIVERSITÄT HAMBURG

DEPARTMENT OF COMPUTER SCIENCE
CHAIR OF THEORETICAL FOUNDATIONS OF COMPUTER
SCIENCE

BETREUT DURCH DANIEL MOLDT

Zusammenfassung

Die Digitalisierung innerhalb der Gesellschaft durchdringt jeden Bereich unseres Lebens. Vom Web, Smartphones, *Augmented Reality* bis zu *Smart Cities* wird die Entwicklung und Beherrschung komplexer Softwaresysteme eine bedeutende Aufgabe sein. Dementsprechend spielt der Wartungsaufwand, die Erweiterbarkeit und die Flexibilität des Systems eine zentrale Rolle für die Softwareentwicklung. Diesen wichtigen Aufgaben hat sich das Modulsystem von Java gewidmet und ermöglicht das Erstellen von Modulen, die spezifische Aufgaben kapseln und die Komplexität innerhalb einer Applikation bändigt.

RENEW ist ein Petrinetz Simulator, der für die Modellierung von komplexen und verteilten Systemen entwickelt worden ist. Dieser setzt auf eine Plugin-Architektur auf, die mithilfe von Plugins die Applikation mit zusätzlicher Funktionalität anreichern lässt, ohne den Kern zu beeinflussen.

Obwohl die Plugin-Architektur sich langfristig bewährte, ist sie nicht für die modernen Herausforderungen gewappnet, denn das Modulsystem von Java führt ein obligatorisches Konzept der Module ein, das den Betrieb von Altsoftware in absehbarer Zeit aufhebt. Dementsprechend muss RENEW sich dieser Herausforderung stellen und an die neue Umgebung anpassen. Zusätzlich wird Gradle als unterstützendes Werkzeug in die Projektumgebung eingeführt.

Zuerst werden die Grundlagen der Java Laufzeitumgebung erarbeitet. Anschließend werden aufbauende Konzepte der Modularisierung von Java diskutiert, die im Folgenden die Migration von bestehenden Applikationen auf das Modulsystem von Java einleiten. Für die Umsetzung der erarbeiteten Konzepte entsteht eine Spezifikation, die von zwei Prototypen erfüllt wird. Zum Schluss folgt eine Evaluation der gesetzten Ziele.

Das Ergebnis setzt sich aus zwei Migrationsszenarien zusammen: Das erste Szenario deckt eine kontinuierliche Migration ab, die an einem Beispiel schrittweise illustriert wird. Im Gegensatz dazu deckt der nachfolgende Prototyp Probleme auf, die entstehen können, wenn bestehende Systeme auf modularisierte Grundlagen aufsetzten.

Das Modulsystem von Java reduziert die Komplexität von RENEW und unterstützt damit die Wartbarkeit der langlebigen Plugin-Architektur. Darüber hinaus fördert das Modulsystem von Java das parallele und gemeinschaftliche Arbeiten an modularisierten Projekten, die in dem RENEW Kontext eine wichtige Rolle spielen. Mit einem strukturierten Migrationskonzept, kann RENEW in einem kontinuierlichen Prozess komplett auf das Modulsystem übertragen werden.

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

1.1 Verteilte Systeme

Fast alle informationstechnischen Anwendungen und Systeme sind heutzutage verteilt, da die Ansprüche an die Rechenleistung ständig wachsen und mit heutigen Mitteln, durch einen Rechenknoten kaum noch zu bewältigen sind.

Zu den wesentlichen Aufgabenstellungen der verteilten Systeme, gehören Konzepte wie Parallelität, Synchronisation, Kommunikation, Kooperation, Koordination und Sicherheit. Diese Aspekte bringen ein hohes Maß an Komplexität mit sich, denn diese müssen alle zugleich auf eine elastische Menge an Knoten angewandt werden und die gegebenen Aufgaben zielgerichtet ebenso wie ausfallsicher erfüllen. Daher gibt es einen hohen Bedarf nach Ansätzen, die eine sinnvolle, robuste und übersichtliche Organisation der verteilten Systeme umsetzen.

Zu diesem Zeitpunkt werden die meisten verteilten Systeme durch eine Spezifikation definiert, die aus Szenarien und lokale Aktionen besteht. Diese bilden jedoch keine Möglichkeit das gesamte Modell formal einheitlich zu analysieren und darzustellen. Diesem Problem widmen sich die Petrinetz Ansätze, die eine einheitliche Darstellung globaler und lokaler Aspekte sowie die Kommunikation in dem verteilten System versprechen. [?]

1.2 Kontext der Petrinetze

Das Konzept der Petrinetze wurde in der Arbeit von *Carl Petri* beschrieben. Dieses besteht aus Stellen, Marken, Kanten und Transitionen, die nebenläufige und kommunizierende Prozesse darstellen können. Der ursprüngliche

S/T-Netz Formalismus wurde mit der Zeit durch gefärbte Marken erweitert, mit dem Ziel äquivalente Strukturen zusammenzufassen und die darin befindlichen Marken zu typisieren [?]. Da die Struktur des Netzes immer noch stark zusammenhängend ist, bleibt die Organisation des Netzes schwer verständlich für das menschliche Auge.

Demzufolge sollte das Netzes auf logisch zusammenhängende Komponente aufgeteilt werden und trotzdem als ein Ganzes gelten. Diese Anforderung wird von den synchronen Kanälen umgesetzt, indem die Netzkomponenten anstelle der Kanten mit synchronen Kanälen verbunden und gezwungen die mit einander verbunden Transitionen synchron zu schalten [?]. Hiermit ist eine Trennung des Netzes nach ihrer Funktionalität erreicht, die qualitativ anspruchsvolle Modelle komplexer und verteilter Systeme entwerfen lässt.

Obwohl das erweiterte Petrinetz ein anspruchsvolles Modellierungswerkzeug bietet, bleibt das gesamte Netzwerk statisch. Demzufolge wurde der nächste Evolutionsschritt in der Entwicklung der Petrinetze mit dem Referenznetz Formalismus getan. Dieser erlaubt dynamisch und bei Bedarf Netzinstanzen zu erstellen und diese als Marken in einem anderen Netz zu bewegen. Somit kann es mehrerer Instanzen eines Netzes geben, die mit unterschiedlicher Belegung im Petrinetz existieren [?].

1.3 Ein Petrinetz Simulator

RENEW ist ein Petrinetz Simulator, der die oben genannten Petrinetz Formalismen unterstützt. Dieser ist in Java geschrieben und bietet eine Oberfläche zum Zeichnen und einen Simulator zum Ausführen der Netze [?].

Da die ursprüngliche Umsetzung von Olaf Kummer eine empfindliche, monolithische Architektur besaß und viel Fachwissen voraussetzte, wurde diese zu einem Plugin Verband von Jörn Schumacher zu Gunsten der Robustheit und Erweiterung unstrukturiert [?]. Ab diesen Punkt kann RENEW über die Plugin Schnittstellen erweitert werden, ohne dass die existierende Logik beeinflusst wird.

Mit seiner Umsetzung delegierte Jörn Schumacher die Ausführung von Logik an Plugins und erstellte eine zentrale Instanz, die den Lebenszyklus bekannter Plugins verwaltet und koordiniert. Die zentrale Instanz nennt sich Plugin-Manager und kann das Verhalten von RENEW mithilfe der Plugins modifizieren. Der Plugin-Manager baut auf zwei primären Namensräumen auf. Zum einen braucht dieser zusätzlichen Bibliotheken zum Verwalten seiner Umgebung und zum anderen braucht er Plugins, die Funktionalität mit sich bringen [?].

1.4 Motivation

Mit der Plugin Architektur hat RENEW den Lebenszyklus weit überschritten. Dies kann an den Codestellen abgelesen werden, die zum Teil aus der JDK 1.4 Version stammen. Somit entsprechen die erstmaligen Gestaltungsmöglichkeiten, Architekturentscheidungen und ihre Umsetzung, nicht mehr den aktuellen Stand der Technik. Vor allem durch die Einführung des Modulsystems von Java, mit dem der JDK sowie der darauf aufbauende Code modularisiert wird, ist der Betrieb von RENEW in der Zukunft gefährdet. Im Zuge dessen ist das Portieren der Applikation unvermeidlich und trägt ein gewisses Risiko mit sich.

1.5 Fragestellung

Es ist unklar wie sich die benutzerdefinierten Namensräume und die so gut wie unberührten Kern-Plugins auf die neuen Modulstruktur übertragen lassen. Zumal die Suche nach dem zusätzlichen Plugin Code eine zentrale Funktion in System darstellt.

Als eine weitere mögliche Schwachstelle der RENEW Applikation, kann die Build-Umgebung von RENEW oder schwammige Software Entwicklung von nicht erfahrenen Studenten auftreten. Zumal die Build-Umgebung zusammen mit RENEW in die Jahre gekommen ist und bestimmte obligatorische Funktionen der modernen Software Entwicklung nicht besitzt und die wilde Kopplung der Plugins untereinander zu Problemen in der Verständlichkeit der Plugin-Abhängigkeiten führt.

Beifolgend stellt sich die Frage: Wie portierbar ist RENEW und was muss beachtet werden, damit der Umstieg auf das Modulsystem von Java gelingt.

1.6 Ziel

Das Ziel dieser Arbeit ist die Anpassung von RENEW an die neuen Anforderungen der Java Laufzeitumgebung, sodass eine minimal und anschließend eine erweiterte Version von RENEW entsteht, die schlussendlich ein Teil des MULAN Rahmenwerks unterstützt. Dafür wird eine moderne Build-Umgebung eingerichtet, die das Modulsystem unterstützt, um anschließend eine modulare RENEW Version aufzusetzen. Der Zweck dieser Arbeit ist eine Struktur, die einen modernen und nachhaltigen Basis der RENEW Applikation anfertigt, um den Entwickler einen unkomplizierten Einstieg in die existierende Logik zu ermöglichen, sowie die Entwickler-Fähigkeiten in der Forschung verteilter Systeme voranzutreiben.

1.7 Aufbau der Arbeit

Die Abschlussarbeit beginnt mit der Einleitung der Grundlagen, die das Arbeiten mit Java näherbringt. Dazu gehören Konzepte wie der Klassenpfad, Klassenlader und Reflektion, die in der Umsetzung von RENEW eine wichtige Rolle spielen. Anschließend werden aufbauende Entwurfsmuster und Prozeduren des Modulsystem von Java erläutert, die das bestehende Java System aktualisieren und neue Konzepte integrieren. Zum Schluss der Grundlagen werden Migrationsansätze vorgestellt, die gängige Migrationsarten abdecken und eine Übersicht über die möglichen Migrationsszenarien vorstellen.

Das nachfolgende Kapitel der Ausgangssituation beschäftigt sich mit der Motivation der Arbeit, den daraus folgenden Zielen sowie Rahmenbedingungen. Zusätzlich wird in diesem Kapitel der aktuelle Zustand der RENEW Applikation erarbeitet, der als Basis für die Prototypen dienen soll.

Als Teil der Umsetzung entstehen zwei Prototypen, die ausgewählte Teile von RENEW modular restrukturieren. Demzufolge wird die Codebasis sowie Design Entscheidungen bei Bedarf modernisiert und auf das Modulsystem von Java angepasst. Dabei soll der Schwerpunkt dieser Arbeit beim Erarbeiten einer modularen RENEW Umsetzung liegen und demnach Plugins in erweiterbare Java Module transformieren.

Dafür wird die bestehende Projekt Konstruktion reorganisiert, sodass Plugins durch mehrere Module repräsentiert wenden können. Darauf aufbauend müssen Plugins die Moduleigenschaften einhalten und die entsprechenden Regeln und Konfigurationen erfüllen. Und zum Schluss wird die RENEW Applikation mit einem modernen *build* Werkzeug organisiert und für die Ausführung verpackt.

Das Ergebnis soll das Modularisierungskonzept in jedem Abschnitt des Lebenszyklus von RENEW unterstützen. Daher wird erwartet, dass das modulare Denken in der Entwicklungsumgebung beim Entwickeln, beim Kompilieren und Verpacken sowie beim Ausführen der Applikation integriert wird.

Das Ergebnis soll als Grundlage für das nächste Lebenszyklus von RENEW dienen und die Entwicklung einfach, flexibel und erweiterbar gestalten. Dafür wir die Unterstützung von MULAN benötigt, welches die wesentliche Funktion von RENEW voraussetzt.

Kapitel 2

Grundlagen

2.1 Java Virtual Machine

Gemessen am Interesse der Anwender und an ihrer Verbreitung ist Java die erfolgreichste Programmiersprache der letzten Jahre. Der Erfolg kam mit der Objektorientierung sowie der Plattformunabhängigkeit. Diese Fähigkeiten brachten eine große und fähige Kommune zusammen, die sowohl aus der Wirtschaft als auch aus dem Forschungsbereich stammt. Dementsprechend ist Java im Laufe der Zeit durch Designmuster, Architekturkonzepte, Paradigmen und aktuellen Sicherheit- sowie Industriestandards erweitert worden. Da RENEW mithilfe der Java Plattform umgesetzt wurde, kann sie sich alle gegebenen Vorteile zunutze machen.

Einer der wichtigen Bausteine von Java, ist die virtuelle Maschine, die das Suchen, Laden und Ausführen einer Codebasis auf allen gängigen Betriebssystemen erlaubt. Demzufolge spielt das Laden von Klassen aus örtlich unabhängigen Komponenten eine große Rolle für die entstandene RENEW Architektur, denn die Plugins müssen gefunden, geladen und kommunikationsfähig eingerichtet werden, sodass sie sich gegenseitig nutzen und beeinflussen können.

In diesem Kapitel werden grundlegende Konzepte des *Klassenpfads*, *Klassenladers* und *Refektion* erläutert, mithilfe dessen die RENEW Plugin-Architektur umgesetzt wurde.

2.2 Klassenpfad

Jede Java-Anwendung wird zuerst in einer für menschlich verständlichen Sprache geschrieben und anschließend in Byte-Code übersetzt. Infolgedessen ist der Code einsatzbereit für die Ausführung und wird an die virtuelle Maschine weiter gereicht.

Um die kompilierten Klassen zu laden, wird von der virtuellen Maschine Ortsangaben mit entsprechendem Code erwartet. Die Ortsangaben nennt man *ClassPath* oder Klassenpfad. Dieser beschreibt eine Liste von Orten, an denen sich die zur Ausführung benötigten Klassen befinden, wie zum Beispiel das lokalen Dateisystem, das Netzwerk oder sogar die Datenbank.

Nachdem der Klassenpfad für die entsprechenden Klassenlader gesetzt ist, kann das *Klassenlader System* die gewünschten Klassen erfassen und in die virtuelle Maschine laden.

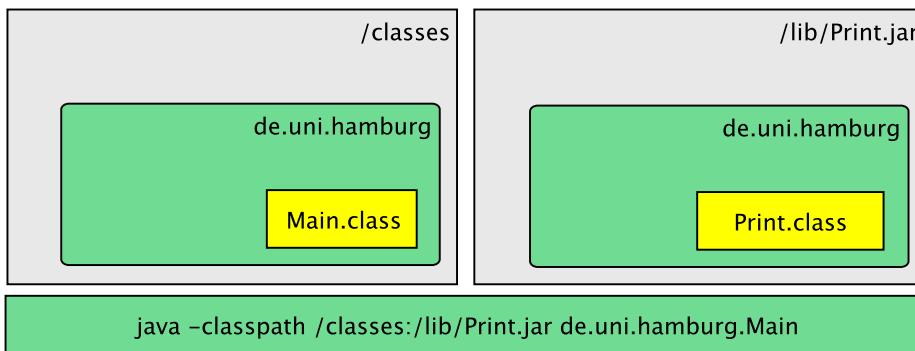


Abbildung 2.1: Java ClassPath

Im Beispiel ?? besteht der Klassenpfad aus einem Verzeichnis sowie einem JAR Archive, die für die Ausführung nötige Klassen umfassen. Da beide Orte eine Dateistruktur beinhalten, unterliegen sie einer Einschränkung: beide müssen die Paketstruktur der Java Klassen widerspiegeln, so dass der *Applikation Klassenlader* diese durchsuchen kann. Abschließend braucht Java einen Startpunkt, mit dem die Applikation ihre Ausführung beginnt.

Beim Starten der Applikation werden Klassen instanziert, indem der Klassenpfad, von links nach rechts, nach dem benötigten Typ durchsucht und diese erstellt. Somit hat der Klassenpfad eine interne Ordnung und eine Abarbeitungsreihenfolge.

Im Beispiel ?? wurde explizit ein Applikationsklassenpfad gesetzt, der für die Ausführung benötigten Klassen zuständig ist. Für den Ablauf großer Applikationen mit viele Abhängigkeiten kann dieser ausgedehnt und chaotisch werden. Von daher bietet Java eine Archivstruktur, die einen standardisierten Aufbau sowie zusätzliche Meta-Information über den Container in sich trägt.

Dank der Java Strukturrichtlinie, befindet sich der komplette Inhalt eines Archivs auf dem Applikationsklassenpfad und kann zusätzlich in der *manifest.mf* Datei erweitert werden. Die *manifest.mf* spielt eine große Rolle in der Entwicklung von Java Applikation, diese kann den Namen, die Version, den Entwickler und die Sicherheitsattribute tragen, die während der Laufzeit ausgewertet werden können. Zum Beispiel wird in ?? der Klassenpfad

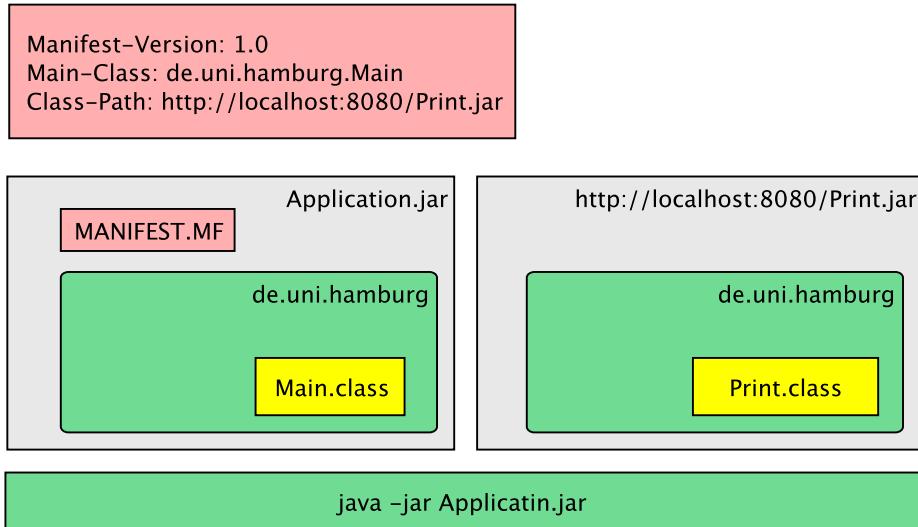


Abbildung 2.2: Jar ClassPath

durch ein Archiv aus dem Web erweitert und für die Ausführung genutzt. Des Weiteren hält die *manifest.mf* einen Einstiegspunkt für die Ausführung, der auf eine Klasse mit der *main* Methode verweist. Somit kann die Applikation in einer kurzen und einfachen Form gestartet werden, da der Ausführungskontext durch die Struktur und die mitgelieferte Meta-Information komplett ist. [?]

2.3 Klassenlader

In den vorherigen Beispielen [??, ??] wurde die Bedeutung und die Rolle des Klassenpfads für die Applikation beschrieben, dennoch muss dieser zuerst verarbeitet werden. Diese Aufgabe wird von dem Klassenlader übernommen, der eine zentrale Rolle in jeder Applikation spielt, weil er nach benötigten Java Klassen für die Instanziierung der entsprechenden Typen sucht. Da es eine wichtige Aufgabe ist, wird die Verantwortung für das Laden der Klassen über eine Menge von Klassenlader aus dem *Klassenlader System* aufgeteilt.

2.3.1 Klassenlader System

Das *Klassenlader System* besteht aus drei integrierten Klassenlader, von denen jeder einen anderen Gültigkeitsbereich für das Laden der Klassen besitzt. Beim Abstieg der Hierarchie wird der Umfang der verfügbaren Quellen breiter und weniger vertrauenswürdig.

Oben in der Hierarchie befindet sich der *Bootstrap-Klassenlader*. Dieser Klassenlader ist verantwortlich für das Laden der grundlegenden Java Klassenbi-

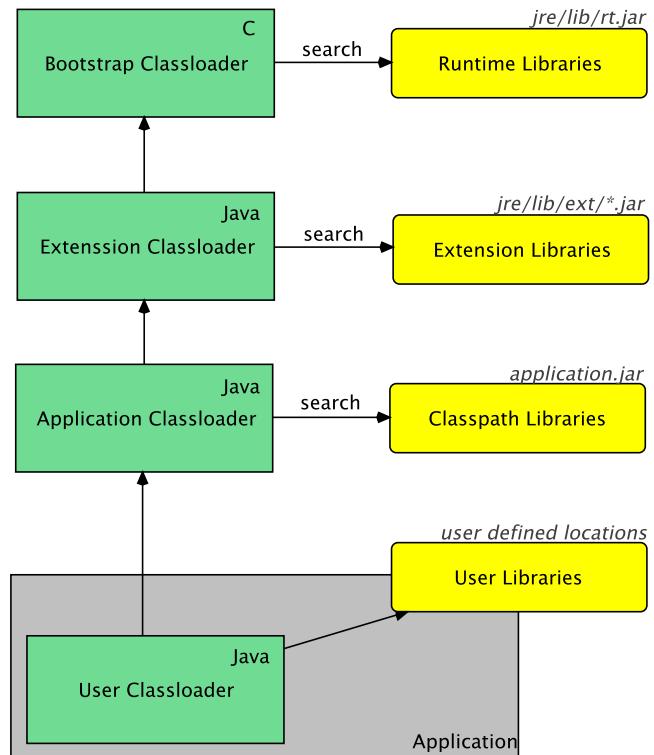


Abbildung 2.3: Klassenlader System [?]

bibliothek, wie zum Beispiel Java-Core-API aus der *rt.jar*. Diese Klassen sind am vertrauenswürdigsten und werden zum Starten der virtuellen Maschine verwendet. Der Klassenlader für Erweiterungen kann Klassen aus dem Standarderweiterungspaket im Erweiterungsverzeichnis *lib/ext* dazu laden. Diese können Java-UI wie kryptografische Erweiterungen beinhalten. Der darunter liegende *Applikation Klassenlader* ist zuständig für unseren Code und lädt Klassen aus dem allgemeinen Klassenpfad einschließlich der zu startenden Anwendung. Zuletzt können benutzerdefinierte Klassenlader erstellt werden, die sich auf der unteren Ebene der Klassenlader-Hierarchie befinden und auf Drittanbieter Bibliotheken zugreifen können. Demzufolge sind diese Quellen nicht sicher genug, um eine große Priorität zuzuweisen, wie zum Beispiel den geladenen Klassen des *Bootstrap-Klassenlader*.

Das in ?? abgebildete Klassenlader System verhindert, dass der Code aus weniger sicheren Quellen vertrauenswürdige Core-API-Klassen ersetzt, indem dieselbe Name als Teil der Core-API angenommen wird. Daraus folgt ein Delegierungsmodell, welches eindeutige Klassen garantiert, da die Klassen-suche von Oben nach Unten der Klassenlader-Hierarchie abgearbeitet wird. [?]

2.3.2 Delegierungsmodell

Das *Klassenlader System* delegiert jede Anfrage zum Laden einer bestimmten Klasse zuerst an seinen übergeordneten Klassenlader, bevor der angeforderte Klassenlader versucht die Klasse selbst zu laden. Jeder Klassenlader hält somit einen Verweis auf einen übergeordneten Klassenlader und ist Teil eines Klassenlader Baums mit dem *Bootstrap-Klassenlader* an der Wurzel.

Wenn eine Instanz einer bestimmten Klasse benötigt wird, prüft der Klassenlader, der die Anfrage bearbeitet, normalerweise mit seinem übergeordneten Klassenlader vorab. Der übergeordnete Klassenlader durchläuft wiederum den gleichen Prozess bis die Delegierungskette den *Bootstrap-Klassenlader* erreicht. Sobald der *Bootstrap-Klassenlader* erreicht wurde, beginnt die tatsächliche Suche nach der gewünschten Klasse.

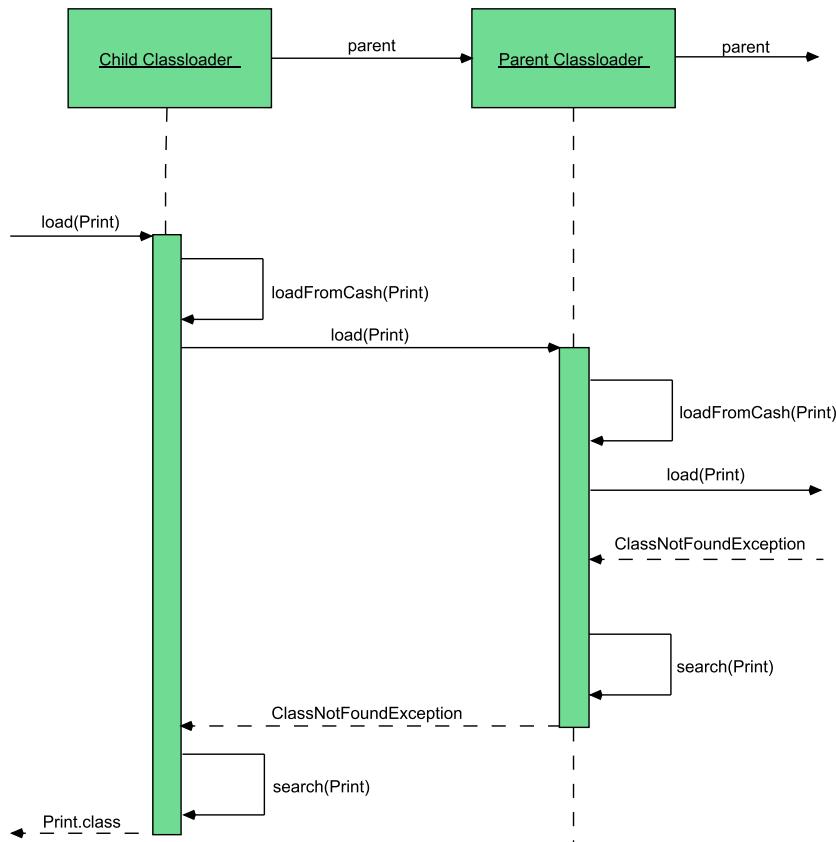


Abbildung 2.4: Klassensuche [?]

Wenn während der Suche ein übergeordneter Knoten eine bestimmte Klasse findet, dann wird diese Klasse die Baumhierarchie herunter zu der Anfrage delegiert. Andernfalls versucht der zuständige Klassenlader als letzter die Klasse selbstständig zu laden. Dies bedeutet, dass eine Klasse normalerweise

nicht nur in dem Klassenlader sichtbar ist, der sie geladen hat, sondern auch für alle untergeordneten Instanzen. Dies bedeutet auch, wenn eine Klasse von mehr als einem Klassenlader in einem Baum geladen werden kann, wird immer die Klasse des übergeordneten Klassenlader eingelesen priorisiert. Dennoch wird vor jedem Laden der Klasse der Cash-Speicher des Klassenladers nach der gewünschten Instanz durchsucht. Wenn diese existiert, wurde die Suche bereits zuvor durchgeführt und keiner der übergeordneten Klassenlader außer dem jetzigen war fähig die Anfrage zu beantworten. Somit kann die Suche beschleunigt werden, indem der Typ sofort zurückgegeben wird. [?]

2.3.3 Namensräume

Geladene Klassen werden sowohl durch den Klassennamen als auch durch den Klassenlader eindeutig identifiziert. Demzufolge werden geladene Klassen in *Namensräume* unterteilt, die vom *Klassenlader System* individuell behandelt werden [?].

Ein *Namensraum* ist eine Gruppe von Klassennamen, die von einem bestimmten Klassenlader geladen worden sind. Wenn ein Eintrag für eine Klasse einem *Namensraum* hinzugefügt wurde, ist es nicht möglich, eine andere Klasse mit demselben Namen und unterschiedlichen Inhalt in den gleichen *Namensraum* einzubinden. Nichtsdestotrotz können mehrere Kopien einer beliebigen Klasse in die Applikation geladen werden, indem für jede Klasse ein Klassenlader mit dem separaten *Namensraum* erstellt wird [?].

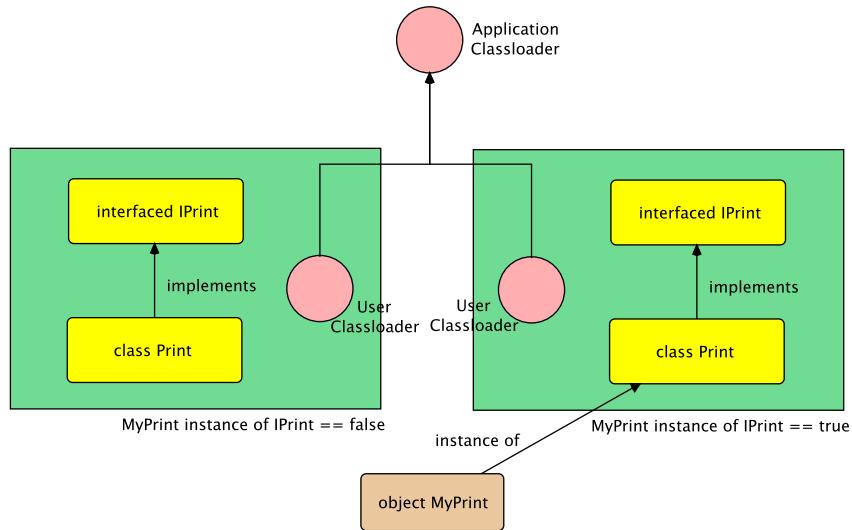


Abbildung 2.5: Namensräume [?]

Die Abbildung ?? zeigt ein Beispiel für eine Klassenidentitätskrise, die sich

ergibt, wenn eine Schnittstelle und die zugehörige Implementierung jeweils von zwei separaten Klassenlader geladen werden. Obwohl die Namen und binären Implementierungen der Schnittstellen und Klassen gleich sind, kann eine Instanz der Klasse von einem Klassenlader nicht als Implementierung des Interfaces von dem anderen Klassenlader erkannt werden. Bei Wunsch kann dieser Umstand gelöst werden, indem das Interface eine Ebene höher rutscht und von den Applikation Klassenlader geladen wird. Somit implementieren beide *Print* Klassen dieselbe Schnittstelle.

Der Klassen Namensraum bietet zusätzliche Sicherheitsfunktionen wie die Kapselung privat deklarierter Pakete. Denn die Namensräume verhindern, dass der weniger vertrauenswürdige Code, der aus der Applikation oder benutzerdefinierte Klassenlader geladen worden ist, direkt mit mehr vertrauenswürdigen Klassen interagieren kann. Beispielsweise wird die Kern-API vom *Bootstrap-Klassenlader* geladen, diese kann *package private* Code enthalten, der bei Anfrage nicht an die unterliegende Klassenlader weitergereicht wird. Auch wenn ein untergeordneter Klassenlader die Paketstruktur der Core-API nachahmt, wird diese nicht als Teil der Java Core-API anerkannt, da dieser von den falschen Klassenlader geladen wurde. Somit verhindert die Verwendung von Namensräumen die Möglichkeit spezielle Zugriffsberechtigungen auf private Pakete zu erhalten, indem man selbst geschriebenen Code diesen zuweist. [?]

2.4 Schnittstellen

Die Schnittstelle und dessen Implementierung spielt eine entscheidende Rolle für das Nutzen der Klassenfähigkeit. Eine Schnittstelle ist ein Vertrag, die die Funktionalität aller Klassen, die dieses implementieren beschreibt. Wenn eine Klasse eine bestimmte Schnittstelle implementiert, verspricht sie die Umsetzung aller in der Schnittstelle deklarierten Methoden.

Somit wird durch die eigene Umsetzung des Schnittstellenvertrags ein mögliches Verhalten für die Nutzer der Schnittstellenbeschreibung implementiert. Daraus folgt ein Kommunikationsvertrag zwischen zwei Objekten, denn wenn eine Klasse eine Schnittstelle implementiert, implementiert diese alle in dieser Schnittstelle deklarierten Methoden und der Methodenaufruf an dieser Klasse wird garantiert ausgeführt.[?]

Im Beispiel ?? wird der Vorteil des Schnittstellenvertrags demonstriert, der das Ausführen der unbekannten *PrintImpl* Umsetzung, durch eine einfache Schnittstellenbeschreibung *IPrint* garantiert. Solange die Implementation sich auf denselben Klassenpfadhierarchie befindet wie die Schnittstelle, wird diese während der Laufzeit auf Kompatibilität geprüft und angewandt. Somit können dynamische Klassenbindungen während der Laufzeit entstehen und Laufzeitbibliotheken ausgetauscht werden, ohne dass die Applika-

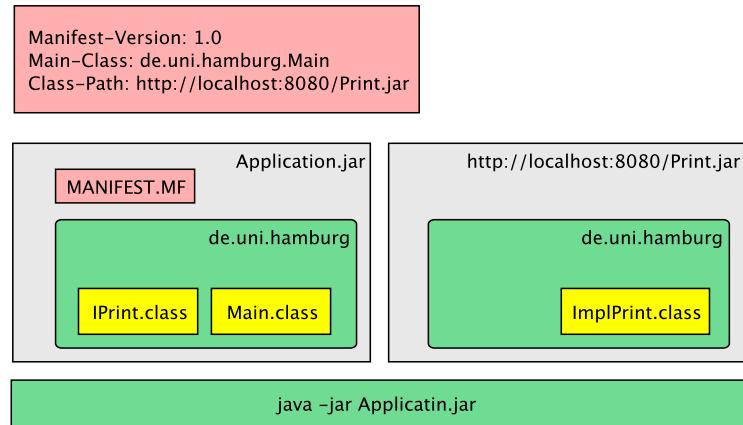


Abbildung 2.6: Schnittstelle

tion verändert wird. Hätte man die Schnittstelle nicht genutzt, würde man die Implementation nur als ein Objekt Type instanziieren können und hätte keinen einfachen Zugriff auf ihre Methoden. In der Konsequenz verbirgt die Schnittstelle ihre Implementierungsdetails der Methoden und gewährt den Vertragspartner keinen Einblick in ihre Umsetzung. Daraus folgt eine einfache Ersetzbarkeit der Implementationsvertreter, ohne den Klienten anpassen zu müssen. [?]

2.5 Reflektion

Reflektion ist die Fähigkeit eines laufenden Programms, sich selbst und seine Softwareumgebung zu analysieren und zu ändern. Somit hat die Applikation eine Möglichkeit, durch Reflexion, die Information über ihre Struktur und ihr Verhalten zu erhalten, um wichtige Entscheidungen zu treffen. Je nachdem welche Information durch die Untersuchung eigener Klassen ausgelesen wurde, können Objekte, die während der Kompilierung nicht präsent waren, mithilfe der Reflektion-API während der Laufzeit instanziert, bearbeitet und genutzt werden. Somit ermöglicht Reflektion das Arbeiten mit Klassen, von denen man im Voraus nicht wissen kann, wie zum Beispiel von Klassen, die in der Zeit nach der Applikation entstanden sind.

In vielen Fällen der Applikationsentwicklung möchte man die Applikation von anderen Nutzern und Entwicklern erweitern lassen, ohne dass diese bei jeder Änderung die komplette Applikation umbauen müssen. Somit stellt sich die Frage, wie erstellt man ein Mechanismus, der mit beliebigen Klassen arbeiten kann.

Man könnte mit den zuvor vorgestellten Schnittstellen- und Implementierungsansatz eine gemeinsame Schnittstelle für Erweiterungen definieren, die

unserer Applikation mit einer Implementation erweitern lässt und die entsprechenden Methoden definiert. Nichtsdestotrotz besteht die Applikation nicht nur aus unserem Code, sondern zusätzlich aus dem Kern und Drittanbieter Bibliotheken, über die wir keine Kontrolle verfügen. Somit ist die Erweiterung der gesamten Codebasis mit der entsprechenden Schnittstelle oder eine Verschachtelung von *instanceof* Blöcken keine simple oder saubere Lösung. Dementsprechend sollte Reflektion genutzt werden. Diese ermöglicht den Einblick in die Klassenstruktur, ohne direkten den Typen zu kennen. Die Klassenstruktur enthalten Informationen über die Klasse selbst, zum Beispiel das Paket, die Superklasse der Klasse sowie die von der Klasse implementierten Schnittstellen. Es enthält auch Details zu den von der Klasse definierten Konstruktoren, Feldern und Methoden.

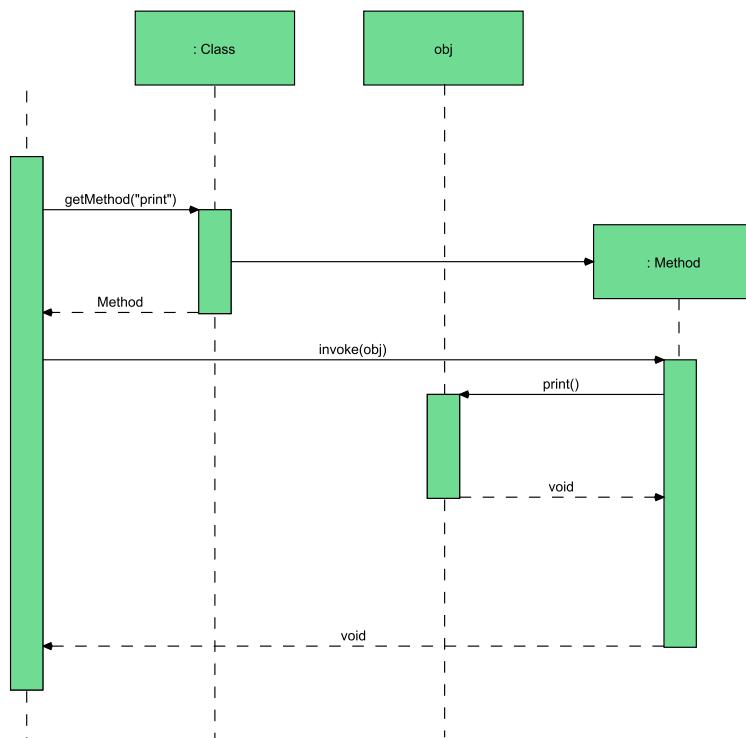


Abbildung 2.7: Aufruf einer Methode [?]

In der Abbildung ?? wird ein Ablauf eines Methodenaufrufs mithilfe von Reflektion visualisiert. Im ersten Schritt muss die Methode gefunden werden. Da wir den Typen nicht kennen und nur für den Stammobjekt Typen deklarieren Methoden nutzen können, lassen wir das Objekt sich selbst inspirieren und die geforderte Methode finden. Dafür wird nach einer bestimmten Methode der Klasse gesucht und bei Erfolg ein Objekt von Typ *Method* zurückgegeben. Das Methodenobjekt enthält die ganze Information über die

gesuchte Methode, wie zum Beispiel Parameter und Rückgabewerte. Ausgerüstet mit der benötigten Information kann die Methode ausgeführt werden. Dafür braucht das Methodenobjekt eine Instanz der passenden Klasse sowie für die Ausführung benötigten Parameter. Nach der Abwicklung wird das Ergebnis der Ausführung vom der Objektinstanz über das Methodenobjekt zurück in den Programmfluss delegiert. [?]

Somit sind die drei Hauptmerkmale einer Klasse ihre Felder, Methoden sowie Konstruktoren durch eine entsprechendes Java Objekt aus der Reflektion-API repräsentiert.

- `java.lang.reflect.Field`
- `java.lang.reflect.Method`
- `java.lang.reflect.Constructor`

Mithilfe des Klassen Typs können die oben genannten Objekte erzeugt und manipuliert werden. Diese bieten Schnittstellen für das Abfragen der Klassenstruktur an und repräsentieren Charakteristika der entsprechenden Klassen.

- `Field[] *.class.getFields();`
- `Method[] *.class.getDeclaredMethods();`
- `Constructor[] *.class.getConstructors();`

Um den Zusammenhang und den Nutzen von Reflektion darzustellen, wird in der Abbildung ?? ein Szenario durchgespielt, das eine unbekannten Typen mithilfe des Konstruktors initialisiert, dessen Methoden aufruft, das Feld bearbeitet und wiedergibt, ohne die Objektstruktur im Voraus zu kennen. Des Weiteren ist zu beachten, dass statische Klassenmethoden sowie private Felder und Methoden mithilfe von Reflektion offen zugänglich gemacht werden können.[?]

```
1  public static void getMethods(@NotNull Class clazz) throws
2      NoSuchMethodException, NoSuchFieldException,
3      InvocationTargetException, InstantiationException,
4      IllegalAccessException {
5     Method method;
6
7     // Instantierung
8     Constructor[] ctors = clazz.getDeclaredConstructors();
9     Object dynamic = ctors[0].newInstance(4);
10    // Aufruf einer privaten Methode
11    method = clazz.getDeclaredMethod("print", String.class);
```

```
12     method.setAccessible(true);
13     method.invoke(dynamic, "Hello World");
14     // Feld Manipulation
15     Field field = clazz.getDeclaredField("version");
16     field.set(dynamic, 5);
17     int version = (int) field.get(dynamic);
18     System.out.println(version);
19 }
```

Listing 2.1: Reflektion in Aktion

Wie in der Abbildung ?? dargestellt ist Reflektion ein mächtiges Werkzeug, das aus der modernen Softwareentwicklung nicht wegzudenken ist und wird in zahlreichen Framework's verwendet, um den Entwickler zu unterstützen.

- Zum Beispiel wird *Dependency Injection* mithilfe von Reflektion realisiert, indem ein Framework, wie zum Beispiel Spring, die entsprechende Implementierung für ein Interface sucht und initiiert. In Diesem Zusammenhang wird Anhand des *implement* Schlüssels und zusätzlicher Meta-Information aus der Klassen *Annotation* ein eindeutiger Kandidat auserwählt und konstruiert.
- Beim Serialisieren und Deserialisieren von Objekten werden die Objektfelder in JSON und wieder zurück konvertiert, ohne die Feldnamen sowie ihre Anzahl zu kennen.
- Die Web-Container Tomcat oder WildFly leiteten die Web-Anfragen an das entsprechende Modul durch das Analysieren der *web.xml* und Anfordern der passenden URI.
- JUnit verwendet Reflektion, um die Methoden einer Klasse nach Test-Annotation zu durchsuchen, um diesen anschließend aufzurufen.

2.6 Gradle

Gradle ist ein Build-Tool ähnlich wie Maven und Ant. Gradle ist das neueste dieser drei Build-Tools, und wird zunehmend eingesetzt. Es ist Open Source und hat viel Akzeptanz bei den Entwicklern gefunden, da es auf die Erfahrung aus den vorhandenen genannten Build-Tools zurückgreift. Mehrere bekannte Projekte wie Android, Spring Framework und Hibernate haben ihre Build-Systeme bereits auf Gradle migriert. Einige der Vorteile, die Gradle gegenüber Maven und Ant hat, sind präzisere Erstellungsskripte und eine flexiblere Erstellungssprache.

Beweggründe für Gradle

Für die Migration von Ant auf Gradle werden im Folgenden die Schwächen von Ant gegenüber Gradle aufgelistet.

- Die Verwendung von XML als Definitionssprache für die Erstellungslogik führt zu übermäßig großen und ausführlichen Erstellungsskripten im Vergleich zu Erstellungswerkzeugen mit einer prägnanteren Definitionssprache. [?]
- Komplexe Erstellungslogik führt zu langen und nicht verwaltbaren Erstellungsskripten. Der Versuch, bedingte Logik wie *if-then*, *for-each* oder *while* Anweisungen mit XML zu definieren, wirkt unnatürlich und aufgeblasen.[?]
- Ant gibt keine Richtlinien zum Einrichten des Projekts. In einem Unternehmen führt dies häufig zu einer Build-Datei, die jedes Mal anders aussieht.[?]
- Gemeinsame Funktionen werden häufig kopiert und eingefügt. Jeder neue Entwickler im Projekt muss die individuelle Struktur eines Builds verstehen.[?]
- Die Verwendung von Ant ohne Ivy erschwert das Verwalten von Abhängigkeiten. In vielen Fällen müssen die JAR-Dateien in die Versionskontrolle eincheckt und deren Organisation manuell verwaltet werden.[?]

Aufbau

Die gradle Umgebung besteht aus mehreren Komponenten. Jede von den Komponenten wird in einem bestimmten Lebenszyklus von Gradle für das Erstellen der Applikation genutzt.

Um Gradle zu starten, werden zwei Ausführbare Skripte mitgeliefert. Für die Ausführung auf dem *Unix* basierte Systeme kann *gradlew* verwendet werden und für die Windows Ausführung ist das *gradlew.bat* Skript zuständig.

Nach dem Anstoßen des Ausführung Skripts, wird Gradle konfiguriert und setzt eine bestimmte Zielversion für die Ausführung fest. Dies geschieht innerhalb der *gradle/wrapper* Ordnerstruktur und initiiert Gradle für die Ausführung. Als nächstes sammelt Gradle die Information über alle Teilnehmer der Projektorganisation und legt eine globale Projektstruktur für die Ausführung fest. Dafür wird in der Initialisierungsphase die *settings.gradle* der Projekte und die Lokale *init.gradle* Konfigurationsdatei ausgelesen. Diese bestimmen Umgebungsvariablen, Eigenschaften und persönlichen Informationen. Des Weiteren werden teilnehmende Projekte gesetzt und instanziert.

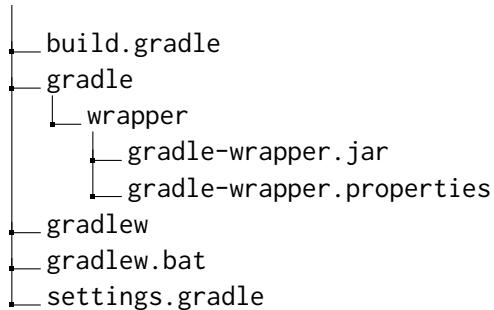


Abbildung 2.8: Gradle Konfiguration

Nachdem die Initialisierungsphase beendet ist, werden erstellte Projekte konfiguriert. Die Konfiguration geschieht laut den *build.gradle* Konfigurationsdatei und bereitet die Projekte für die Ausführung vor. Zum Schluss wird das Bauen durchgeführt, indem eine Reihe von Ausführungsschritten (Tasks) von dem Nutzer aufgerufen werden.[?]

Gradle build Skript

```

1 // Deklaration der genutzten Plugins
2 plugins {
3     id 'java'
4 }
5 // Bibliothek Quellen
6 repositories {
7     mavenCentral()
8 }
9 // Projektabhängigkeiten
10 dependencies {
11     compile('javax:javaee-web-api:8.0')
12 }
13 // Klassenpfade
14 configurations {
15     libs
16     plugins
17     compile
18 }
19 // Felder
20 group 'de.firm'
21 description 'my first application'
22 version 'version'
23 defaultTasks 'war'
24 // Projektstruktur
25 sourceSets {
26     main {
27         java {
28             srcDirs ['src/main/java']
29         }
30     }
31 }
32 }
33 
```

```

30         webAppDirName 'src/main/webapp'
31         outputDir file('out/classes')
32     }
33     resources {
34         srcDirs ['src/main/resources']
35         output.resourcesDir = file('out/resources')
36     }
37 }
38 // Konfiguration eines Abarbeitungsschritts
39 task war {
40     setGroup("gradle")
41     setArchivesBaseName(name)
42     webInf {
43         into('classes') {
44             from sourceSets.main.java.outputDir
45             into('META-INF') {
46                 from(sourceSets.main.resources.files) {
47                     include("persistence.xml")
48                 }
49             }
50         }
51     }
52 }
53 metaInf {
54     from(sourceSets.main.resources) {
55         include("import.sql")
56     }
57     manifest {
58         attributes 'version': war_version
59         attributes 'description': war_description
60         attributes 'creator': war_creator
61         attributes 'classifieire': war_classifieire
62     }
63 }
64 }
65 // Task Graph Manipulation
66 deploy.dependsOn(war)
67

```

Listing 2.2: Gradle in Aktion [?]

In der Abbildung ?? ist eine Standard Struktur eines Gradle Build Skript abgebildet, die mit einem Java Plugin arbeitet und ein war Archiv erstellt.

Kapitel 3

Modularisierung

Modularisierungsansätze finden sich so gut wie in jeder Software wieder, da es sich um ein grundlegendes Prinzip für die Beherrschung eines Systems handelt. Gerade in der Java-Welt wird seit jeher das Ideal von lose gekoppelten Systemen verfolgt, denn diese generieren die Struktur in großen Softwareprojekten, indem das Gesamtprodukt in kleine, praktische Bestandteile zerlegt wird.

Die Entwicklung von kleinen Projekten mit einer übersichtlichen Codebasis ist einfach zu überblicken und braucht keine strukturelle Unterstützung, um den Entwickler Architektur und Funktion darzustellen. Dennoch ist die Zukunft eines Projekts nicht immer eindeutig und kann mit der Zeit an Größe und Komplexität gewinnen. Mit der Größe des Projektes, wächst der Geschäftskontext und damit die Zahl der beteiligten Personen. Diese repräsentieren verzwickte Wünsche und Ziele, die an einer Stelle im Projekt nicht sauber umsetzbar sind. Infolge dessen ist die richtige Aufstellung eines Projektes von Grund auf eine zukunftssichere Entscheidung.

Ohne die Modularisierung werden Änderungen an großen Projekten mühselig und mit unerwarteten Nebeneffekten umgesetzt. Sowohl das Bauen und Ausrollen des Projekts als auch der Betrieb der Applikation ist eine lange und aufwendige Aufgabe, die mit jedem kleinen Fehler die komplett Applikation Neustarten lässt oder das Ausrollen unterbricht. Somit können kleine Fehler das ganze Produkt aus dem Gleichgewicht bringen. Aus diesem Grund sollen Module diese Probleme adressieren und die Applikation in autonome, kleine Einheiten aufteilen, die unabhängig voneinander ihre Funktionalität anbieten.

3.1 Ziele der Modularisierung

Die Modularisierung beschäftigt sich mit der Aufteilung eines Systems in Module, die Komplexität verringern, indem die einzelnen Module getrennt voneinander betrachtet und verstanden werden. Dies wiederum unterstützt die Wartbarkeit der einzelnen Module. Darüber hinaus vereinfachen die von der Modularität geforderte Schnittstellen Spezifizierung die Kommunikation zwischen den Modulen und fördert damit die Erweiterbarkeit des Systems. Da die Module austauschbar sind und unabhängig voneinander betrieben werden können, eröffnen sich neue Möglichkeiten der Softwareentwicklung. Wie zum Beispiel die Erstellung verschiedenen Varianten der Umsetzung, durch die Rekombination existierender Module, ohne die ganze Applikation zu in Betracht zu ziehen.[?, ?, ?]

Um die Aufgabe des Java Modularisierung zu verstehen, bedarf es einer Aufstellung von Zielen und Qualitäten, denen sich die Modularisierung stellt. Für JPMS sind diese eindeutig in der *JSR 376* [?, ?] beschrieben und spezifizieren die Folgenden Qualitäten.

Kapselung

Die Kapselung beschreibt ein Kontrollmechanismus, der die interne Struktur eines Moduls verwaltet. Demzufolge hat das Modul die komplette Kontrolle über ihre interne Struktur und kennt die Zugriffsrechte ihrer Bestandteile, indem das Modul die Zugriffsrechte ihrer inneren Struktur explizit deklariert.

Interoperabilität

Die Interoperabilität beschreibt die Kommunikationsfähigkeit der Software mit anderen diversen Systemen, unabhängig von ihrer Sprache oder Plattform, mit der diese betrieben wird. Darum bieten Module Schnittstellen an, mit denen sie Dienste anbieten und anfordern können.

Zusammensetzbarkeit

Aus der Interoperabilität geht die Zusammensetzbarkeit hervor. Diese steht für die Wiederverwendbarkeit der in sich abgeschlossenen Module für unterschiedliche Zwecke in unterschiedlichen Systemen, indem man diese auf bestimmte Art und Weise kombiniert.

Erweiterbarkeit

Die Erweiterbarkeit hilft den modularen und zusammengesetzten Systems ihre Funktionalität zu skalieren, indem die Software durch individuelle Einheiten ergänzt werden kann.

Autonomie

Mit der Autonomie werden unnötige Abhängigkeiten aufgelöst und nur die nötige Funktionalität für die entsprechende Aufgabe in einem Modul abgelegt. Somit können einzelne Module im Betrieb auch dann bleiben, wenn Teile des Systems nicht reagieren.

3.2 Modulstruktur

Die zuvor aufgeführten Ziele der Modularisierung liefern bereits eine Vorstellung der Modulanforderungen. Primär erfüllt ein Modul einen abgeschlos-

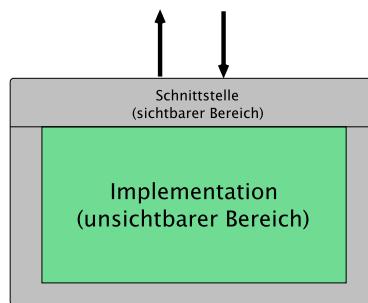


Abbildung 3.1: Simple Modulstruktur [?]

senen Aufgabenbereich und beinhaltet die dafür nötigen öffentlichen sowie privaten Operationen und Datenfelder. Die Kommunikation eines Moduls mit anderen Modulen und der Außenwelt erfolgt über eindeutig spezifizierte Schnittstellen.

Somit dient das Modul als ein Behälter für Objekte, der aus einem unsichtbaren und einem sichtbaren Bereich besteht. Der sichtbare Bereich ist die Schnittstelle des Moduls und ist die Aufzählung derer Objekte, die das Modul nach außen hin zur Verfügung stellt. Der Zugriff auf diese erfolgt über definierte Operationen in der Modulschnittstelle. Der unsichtbare Teil beherbergt die eigentliche Implementierung, also die umgesetzten Operationen und Daten. Unter diesen Umständen reduziert sich die Komplexität des Moduls für den Nutzer von der gesamten Implementation auf die Schnittstellen. [?, ?, ?, ?]

In der Abbildung ?? wird die interne Struktur sowie entsprechenden Verbindungen eines Moduls genau betrachtet. Zu sehen sind drei Module, die ihre Dienste mit dem *exports* Schlüssel über die Schnittstellen anbieten und diese bei Bedarf mit anderen Modulen kombinieren können, indem weitere Funktionalität durch den *requires* Schlüssel von zusätzlichen Modulen angefordert wird. Die interne Umsetzung der Funktionalität bleibt jedoch verborgen und kann Modulübergreifend nicht nachverfolgt werden.

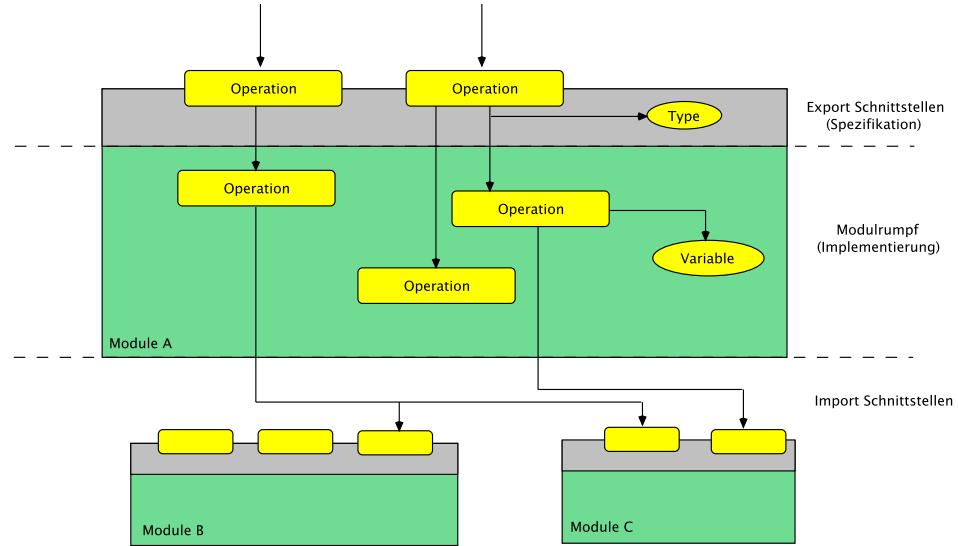


Abbildung 3.2: Schematischer Aufbau eines Moduls [?]

3.3 Moduleigenschaften

Modul Definition: Ein Modul ist eine Sammlung von Algorithmen und Daten bzw. Datenstrukturen zur Bearbeitung einer in sich abgeschlossenen Aufgabe. Die Verwendung des Moduls (d.h. seine Integration in ein Programm-System) erfordert keine Kenntnis seines inneren Aufbaus und der konkreten Realisierung der gekapselten Algorithmen und Daten(-strukturen). Seine Korrektheit ist ohne Kenntnis seiner Einbettung in ein bestimmtes Programmssystem nachprüfbar. [?]

Aus dieser Definition können folgende Eigenschaften abgeleitet werden, die ein Software-Modul beschreiben:

- Zusammenfassung von Operationen und Daten zur Realisierung einer in sich abgeschlossenen Aufgabe
- Kommunikation mit der Außenwelt nur über eine eindeutig spezifizierte Schnittstelle
- Nutzung des Moduls möglich ohne Kenntnis des inneren Ablaufs
- Die Struktur jedes Moduls sollte einfach genug sein, um vollständig verstanden zu werden.
- Anpassungen eines Moduls sollte ohne Kenntnis der Implementierung sowie ohne Einfluss auf das Verhalten anderer Module durchführbar sein.

- Korrektheit des Moduls durch Tests nachprüfbar ohne Kenntnis seiner Einbettung
- Wiederverwendbarkeit der Funktionalität im anderen Kontext

3.4 Modulentwurfskriterien

Nachdem die Struktur des Moduls klar bestimmt wurde, muss die Umsetzung einer Applikation mit Modulen auf Qualitätsmerkmale abgeglichen werden. Da die Aufteilung eines Entwurfsproblems in kleinere Teilprobleme nicht selbstverständlich ist, kann diese mit verschiedenen Techniken und auf diverse Weise umgesetzt werden und bietet daher keine Garantie eines sauberen Entwurfs. Die Kunst Funktionalität in einem einzelnen Modul zu kapseln und diese mit geringer Abhängigkeit vom Restsystem betreiben zu können, kann mithilfe bestimmter Kriterien bewertet und angepasst werden. [?, ?, ?, ?, ?, ?]

Bei der Modularisierung sind folgende Entwurfskriterien zu berücksichtigen:

- Modulgeschlossenheit
- Maximale Modulbindung
- Minimale Modulkopplung
- Minimale Schnittstelle
- Modulanzahl
- Modulgröße
- Testbarkeit
- Seiteneffektfreiheit
- Importzahl
- Modulhierarchie

Mithilfe der *Modulgeschlossenheit* wird die Abhängigkeit des Moduls von anderen Modulen reduziert und lässt diese separat bearbeiten und austauschen. Somit kapselt ein Modul eine bestimmte Funktionalität, die von Anfang bis zum Ende intern verarbeitet werden kann. Direkt daraus folgt im besten Fall eine *maximale Bindung* oder starker Zusammenhang innerhalb eines Moduls, indem die internen Komponenten bestens mit einander verzahnt sind und sich gemeinsam mit einer gezielten Aufgabe beschäftigen. In der Konsequenz entsteht ein eingeschränkter Wartungsraum für Entwickler, die sich mit der entsprechenden Funktion beschäftigen. Um die Bindung der

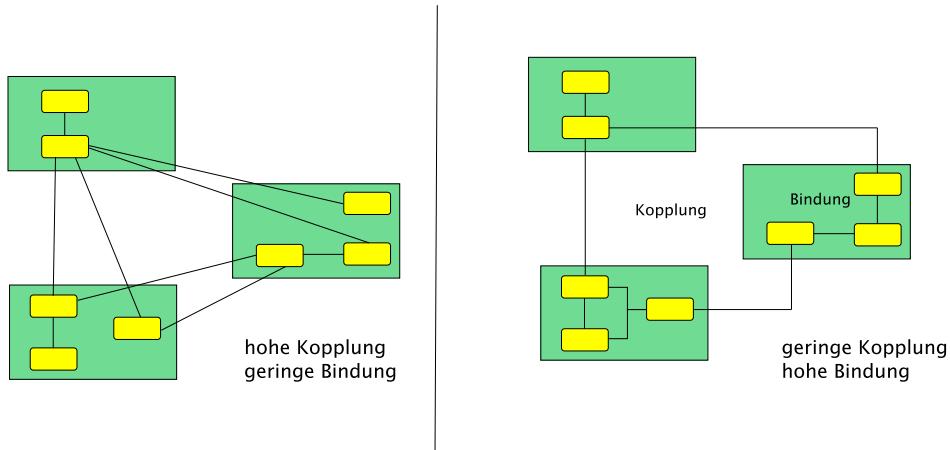


Abbildung 3.3: Modulbindung und Modulkopplung [?]

Komponenten innerhalb eines Moduls zu messen, können die Abhängigkeiten in verschiedenen Kategorien eingeteilt werden: logisch, zeitlich, prozedural, sequentiell, informal und funktional.

Komplementär zu der *maximalen Bindung* beschreibt die *minimale Kopplung* die Anzahl der Verbindung zwischen den Modulen. Diese sollte natürlich klein gehalten werden, um die Abhängigkeit zu reduzieren. Die *minimale Kopplung* hat somit einen direkten und positiven Einfluss auf die Anzahl der Schnittstellen, indem diese übersichtlich und eindeutig die Funktion des Moduls beschreiben. Andernfalls kann eine starke Kopplung die Komplexität heben und Fehler begünstigen, indem der Umfang an Daten, die zwischen den Modulen ausgetauscht werden, erhöht wird. Eine *minimale Kopplung* ist ein guter Ansatz den unnötigen Datentransfer zu reduzieren, garantiert aber keine lose Kopplung von den umgebenen Modulen. Daher sollte der Begriff *Seiteneffektfrei* eingeführt werden. Dieser beschreibt den Einfluss eines Moduls auf seine Umgebung, indem das Modul Unverzichtbar für die Gesamtfunktionalität wird und der Austausch die Anpassung verknüpfter Module nach sich zieht. Das ist öfters der Fall, wenn eine Aufgabe modulübergreifend gelöst werden muss und die Aufgabenkapselung für diesen Zweck aufgelöst wird.[?, ?, ?, ?, ?]

3.5 Modulararten

Das Modulsystem von Java unterscheidet die Module in fünf unterschiedliche Arten, diese richten sich nach der Aufgabe und ihrer Umsetzungsstruktur. Zum einen gibt es die JDK *Plattform Module*, die die Kernfunktionalität der Java Laufzeitumgebung bieten und Pakete wie *java.lang*, *java.io* und *java.net* mit sich bringen. Andererseits gibt es die Benutzer konstruierten

Applikationsmodule, die durch eine explizite Komposition bestimmte Aufgaben erfüllen. Beide Modultypen beinhalten eine *Modulbeschreibung*, die dessen Abhängigkeiten und Schnittstellen beschreibt.

Obwohl mit den vorher genannten *expliziten Module* Softwaresystemen realisieren lassen, fehlt die Offenheit bestimmter Module oder ihrer Pakete für die Umsetzung der Reflektion Bibliotheken, die dynamischen Zugriff auf unsere Pakete während der Laufzeit benötigen. Wie im Kapitel ?? besprochen ist Reflektion ein wichtiges Werkzeug in der Softwareentwicklung und wird in dem neuen Java Modulsystem unterstützt. Um Reflektion in einem Modul zu aktivieren, reicht es lediglich das ganze Modul als *open module* oder mithilfe des *opens package.name* Schlüssels ein spezielles Paket des Moduls in der *Modulbeschreibung* zu deklarieren. Damit hätte man ein für Reflektion offenes Modul und könnte dessen öffentlichen sowie privaten Klassen dynamisch aus jedem Modul auf dem Modulpfad aufrufen. Da diese Module das Konzept der starken Kapselung aufgeben, werden diese zu einem besonderen Typen der *offenen Module* zugeordnet. [?, ?, ?, ?]

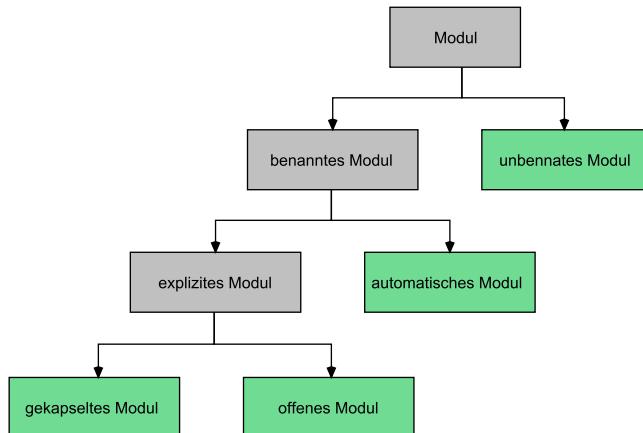


Abbildung 3.4: Modulararten [?]

Die nachfolgenden Modultypen sind Pseudo-Module, die für die Unterstützung der Abwärtskompatibilität eingeführt worden sind. Dementsprechend sollen diese Module eine Brücke zwischen existierender Applikation und der modularisierten Architektur bilden.

Das *unbenannte Modul* beschreibt alle Klassen und JAR's, die sich parallel zu der Codebasis des Modulpfades auf dem Klassenpfad befinden. Das *unbenannte Modul* beschreibt somit den Legacy-Teil der Codebasis, die noch migriert werden muss und es noch nicht tun kann. Daher wird mit der Bezeichnung *unbenanntes Modul* eine Zugriffsbarriere zwischen der modularisierten und der legacy Architektur errichtet, die die *expliziten Module* vom Zugriff auf den veralteten Klassenpfad abgrenzt. Denn dieses trägt keinen

Namen und kann somit vom Entwickler nicht pragmatisch referenziert werden.

In Folge dessen entsteht eine asymmetrische Kommunikation zwischen den Architekturen. Die *expliziten Module* arbeiten nur auf dem Modulpfad im neuen System und das *unbenannte Module* darf zusätzlich zu den klassischen Klassenpfad auf den modernen Modulpfad zugreifen. Diese Umsetzung lässt eine inkrementelle Migration der Codebasis auf das Modulsystem zu und bleibt lauffähig, obwohl die Applikation eine interne Versionsdiskrepanz beinhaltet. [?, ?, ?]

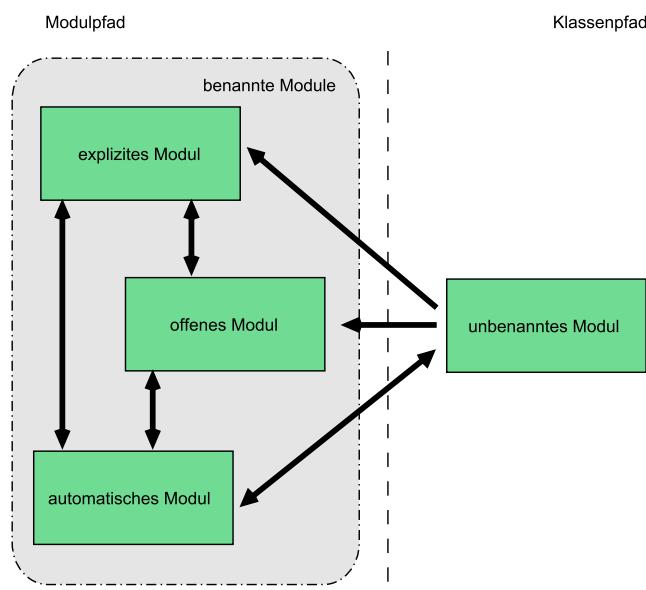


Abbildung 3.5: Modulzugriffsrechte [?]

Das letzte Modul beschreibt ein Modul mit speziellem Verhalt, dass sich zwischen den Architekturen stellt und eine Brücke zwischen den Modulpfad und Klassenpfad errichtet. Die *automatischen Module* beschreiben einen Migrationsansatz der bestehenden Bibliotheken, die vom Klassenpfad auf den Modulpfad verschoben werden und keine *Modulbeschreibung* besitzen. Diese kriegen einen Modulnamen zugewiesen und können über diesen von den *expliziten Modulen* aufgerufen werden. Somit übernimmt Java die Kopplung der *automatischen Module* mit allen *expliziten Modulen*, indem alle internen Pakete für die Nutzung offen gelegt werden und alle Module auf dem Modulpfad für die Verwendung importiert werden. In der Folge ist eine Legacy-Bibliothek auf den Modulpfad funktionstüchtig und bietet eine ganz besondere Fähigkeit, nämlich die wechselseitige Kommunikation zwischen dem Modulpfad sowie dem Klassenpfad. Dank dieser Fähigkeit können Bibliotheken migriert werden und beide Architekturen zu gleich unterstützen. Dieses Ver-

halten fördert die Entwickler ihren Code für den Modulpfad zu entwickeln, da die nötigen Legacy-Bibliothek der Applikation in beiden Architekturen zugleich verfügbar ist.

Dennoch schaffen die *automatischen Module* zusätzliche Komplexität in die Architektur, indem alle Module mit diesem verbunden werden. Daraus folgt eine starke Kopplung und somit eine unübersichtliche, starke Abhängigkeit zwischen den Modulen.[?, ?, ?]

Nichtsdestotrotz bietet das automatische sowie das unbenannte Modul diverse Migrationsszenarien, die flexible Wege für die Modernisierung der Applikation anbieten.

3.6 Modulkopplung

Die Einführung des Modulsystems in Java 9 integriert das Konzept der Aufteilung einer monolithischen Softwareumsetzung in übersichtliche mit einander sachlich verbunden Modulen. Diese Idee wird zuerst von Java selbst umgesetzt, um als Beispiel für den aufbauenden Code zu fungieren. In der praktischen Umsetzung formuliert Java die Modulbeschreibung mithilfe der *module-info.java* Datei, die drei Kopplungstypen enthalten kann: die durch *requires*, *exports* und *opens* Deskriptoren beschrieben wird.

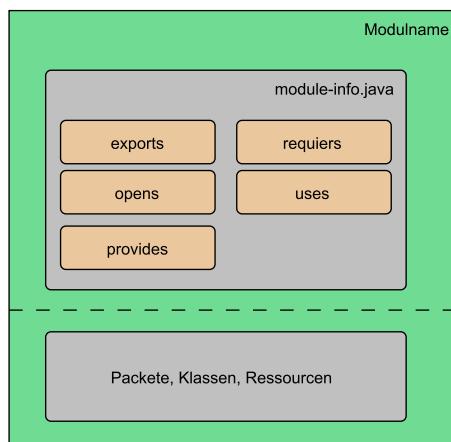


Abbildung 3.6: Die Schnittstellenbeschreibung *module-info.java*

Die in der Abbildung ?? dargestellten und vorher diskutierten Kopplungsarten, können die Zugriffsberechtigungen ferner einschränken. Diese können ihre Schnittstelle exklusiven Modulen öffnen, die fortan als eine transitive Verbindung weiterreichen und obendrein als eine optionale Abhängigkeit deklarieren. Die *uses* und *provides* Schlüssel beschreiben eine Serviceanfrage sowie einen Serviceangebot, die durch den *Java ServiceLoader* mit einander

verknüpft werden.

Der *ServiceLoader* übernimmt in diesem Fall die Rolle des Registrierungsdienstes und vermittelt das Angebot und die Nachfrage nach Funktionalität innerhalb der Applikation. Das Konzept der Dienstregistrierung und der Dienstverwaltung geht über die Grundlagen hinaus und wird hier nicht weiter diskutiert, dessen ungeachtet ist es eine zusätzliche Möglichkeit die Modulkopplung zu minimieren. [?, ?]

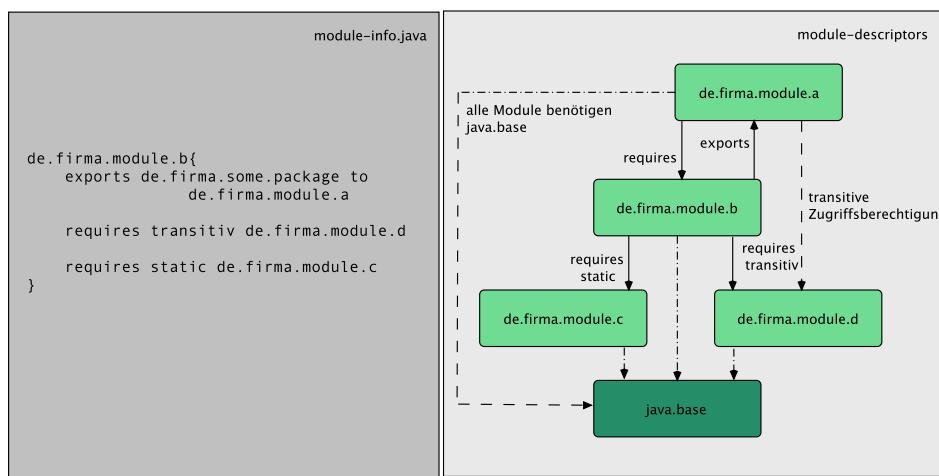


Abbildung 3.7: Abwandlung der Kopplungsarten

Im Folgenden werden die in der Abbildung ?? dargestellten Möglichkeiten der Kopplungstypen gelistet. Zu beachten ist die Wechselbeziehung zwischen den Modulen, die Pakete anbieten und Module anfordern. [?]

requires

requires Modul
requires transitiv Modul
requires static Modul
requires transitiv static Modul

exports

exports Packet
exports Packet *to* Modul-1, Modul-2

opens

opens Packet
opens Packet to Modul-1, Modul-2

uses

uses Service-Schnittstelle

provides

provides Service-Schnittstelle *with* Service-Impl-1, Service-Impl-2

Wie in der grafischen Darstellung ?? abgebildet, handelt es sich bei den Kopplungstypen um Zugriffsrechte, die als ein offener Vertrag zwischen Modulen aufgestellt werden. Dementsprechend dienen die Kopplungsschlüssel nicht nur der Lesbarkeit und Autonomie, sondern erweitern die Prozedur des Klassenladens durch explizite Schnittstellen und Zugriffsberechtigungen.[?]

3.7 Module Classloading

Im Abschnitt ?? wurden Namensräume vorgestellt, die Klassen von einemander trennen und diese als separate Software Komponenten behandeln, um die Sichtbarkeit der Codebasis gegenüber dem Restsystem abzugrenzen. Je doch bring dieses Feature einen großen Aufwand mit sich, denn sofern die Applikation eine große Anzahl an Bibliotheken benutzt und jedes davon auf einem eigenen Klassenlader betreiben möchte, wächst der Wartungsaufwand mit der Anzahl der Bibliotheken.

Mithilfe der Module und dessen neuem Ansatz der internen Kapselung, soll dieses Problem adressiert werden, indem separate Zugriffsräume für jedes Modul innerhalb eines Klassenladers definiert werden, die sicherstellen, dass die interne Struktur eines Moduls während der Laufzeit nicht kompromittiert werden kann.

Für die Garantie der Modulkapselung, wird das im Abschnitt ?? vorgestellte *Java Klassenlader System* nicht ersetzt, sondern mit zusätzlichen Kontrollen versehen, die strikt nach deklarierten Modulbeschreibung Zugriff gewährleistet. Im Falle der Nichteinhaltung der Zugriffsrechte zwischen den Modulen wirft der Klassenlader neue Fehlermeldungen wie die *IllegalAccessException* oder die *IllegalAccessError*. Somit bleibt die ehemalige Klassenlader Hierarchie erhalten, die an das Modulsystem von Java angepasst worden ist. [?, ?]

Um die neuen Kontrollmechanismen im Java Kern zu verankern, wurde der die *rt.jar*, die für die Ausführung von Java Code zuständig ist, auf Module mit expliziten Aufgabenbereichen aufgeteilt. Wie in der Abbildung ?? abgebildet, beschäftigt sich das aktualisierte *Klassenlader System* mit dem Laden bestimmter Module, die in Gültigkeitsbereiche fragmentiert sind. Diese nutzen immer noch das Delegierungsmodell ?? und delegieren jede Anfrage zuerst an den Wurzel-Klassenlader. Mit dem Modulsystem von Java wurde dieses Verfahren durch zwei Sicherheitsverfahren erweitert. Das erste Verfahren löst eins der größten Schwierigkeiten der Java Entwicklung, nämlich das Management der Abhängigkeiten eines Systems. Diese Herausforderung wird als *Jar-Hell* bezeichnet und beschreibt eine komplexe Applikation, die eine lange Liste an genutzten Drittanbieter Bibliotheken benötigt. Das Problem taucht auf, da der Java Code den Klassenpfad nicht kontrolliert und

annehmen muss, dass alle benötigten Bibliotheken auf den Klassenpfad vorhanden sind. Die lokal eingerichtete Maschine oder Entwicklungsumgebung kann das Management verwalten, jedoch kann diese nicht garantieren, dass die Software in einer anderen Umgebung funktionsfähig bleibt.

Dieser Anforderung hat sich das Modulsystem von Java gestellt und bietet die phasenübergreifende Wiedergabetreue auf allen Systemen, indem jedes Modul eine Voraussetzung deklariert, die der Modulpfad erfüllen muss, bevor die Applikation ihre Arbeit beginnt. Somit wird das gleiche Verhalten sowie Fehlerprüfungen in der Kompilierungs- und Ausführungsphase erreicht.

Das zweite Verfahren garantiert die Einhaltung der Nutzungsrechte einer Bibliothek, indem die Nutzung einer Bibliothek über klar definierte Schnittstellen geschieht und unterbindet somit den impliziten Zugriff auf den Inhalt innerhalb eines Namensraums. Beide Verfahren berufen sich auf die *module-info.java* Konfigurationsdatei, die in sich die benötigte Information verankert.

Um die Klassen aus dem Modulpfad zu laden, wurde das *Klassenlader Systems* an den Modulpfad angepasst und lädt jetzt Module, die nach Sicherheitsberechtigungen bestimmten Klassenlader zugewiesen werden. Der *Bootstrap* Klassenlader ist der Klassenlader der JVM und genießt alle Sicherheitsprivilegien. Dieser lädt die *Core Java-SE* und *JDK-Module*, wie *java.base* und *java.logging*. Der Extension Klassenlader wurde von dem Plattform Klassenlader ersetzt und lädt jetzt die *Plattform Java-SE* Module, wie zum Beispiel die *java.sql*, oder die *java.xml.ws* Bibliotheken. Und zuletzt bleibt der Applikation Klassenlader, der unsere Applikationsmodule auf dem Modulpfad verwaltet. [?, ?, ?]

3.8 Modulschichten

Obwohl das Klassenlader-Konzept von Java seine Gültigkeit behält, gibt es mit dem Einzug der modularisierten Plattform einige Änderungen.

Mit dem Modulsystem von Java wird ein neues Konzept der Modulschichten eingeführt, das für das Laden von Java Code während der Laufzeit verantwortlich ist. Die Modulschichten kapseln, validieren und laden bestimmte Modulgruppen, die während der Laufzeit in die Applikation integriert werden können. Dazu zählen explizite, offene sowie unbenannte Module.

Das Laden von zusätzlichem Code war bereits mit dem Klassenlader System möglich indem zusätzliche Klassenlader instanziert und ausgetauscht werden konnten, jedoch besitzen die Klassenlader keine Mechanismen zum Validieren des eingelesenen Codes, sodass unerwartetes Verhalten im Betrieb der Applikation zu jedem Zeitpunkt passieren kann.

Mithilfe der Modulschichten wird das bestehende Klassenlader-Konzept erweitert, indem zusätzliche API's für die Verwaltung von dem dynamischen

Code eingeführt und das zusätzliche Erzwingen der Korrektheit der Modul-abhängigkeiten innerhalb des Systems garantiert wird.

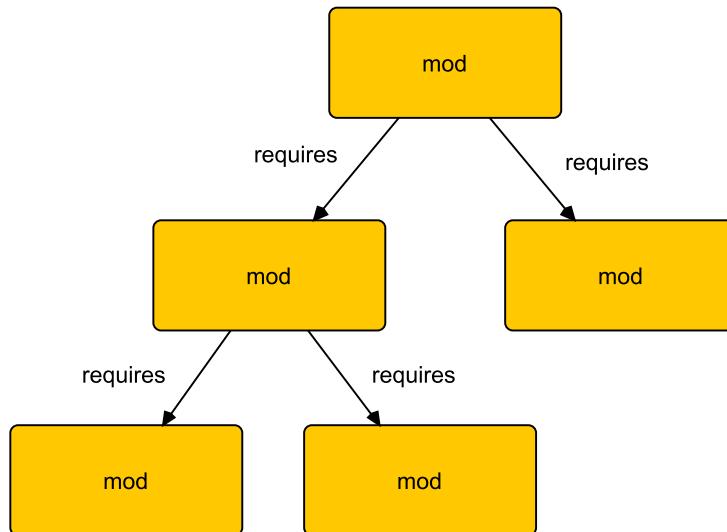


Abbildung 3.8: Modul Graph [?]

Das Laden von zusätzlichen Modulen geschieht in zwei Phasen. Zuerst wird eine Konfiguration erstellt, die ausgewählte Module für das Laden in die Applikation beschreibt. Die Konfiguration benötigt eine Modulmenge sowie die ausgewählte Wurzel-Module, aus den anschließend alle Modulabhängigkeiten ermittelt und validiert werden. Dafür wird ein neu eingeführtes Objekt genutzt, nämlich der *ModulFinder*, der für ein gegebenes Verzeichnis die Meta-Informationen aller Module ausliest und für die Nutzung bereitstellt. Im nächsten Schritt müssen die gefundenen Module auf Konsistenten geprüft werden. Dazu wird ein Modul Graph ?? erzeugt, der aus den gegebenen Wurzel-Modulen alle geforderten Modulbeziehungen auflöst, auf *Zyklen*, *Split Packages* sowie valide Zugriffsrechte inspiziert und im Anschluss den Code für Ausführbar erklärt. Der Prozess stellt sicher, dass alle Abhängigkeiten, einschließlich der indirekten Abhängigkeiten, aufgelöst werden können. Das Ergebnis besteht aus einer betriebsfähigen Konfiguration mit einem Strukturgraphen der aufgelösten Abhängigkeiten, die für die Ausführung im Applikationskontext benötigt werden. [?]

Nachdem die Konfiguration ausgewertet wurde, kann eine Modulschicht erstellt werden, die alle Module der konsistenten Konfiguration instanziert. Dafür wird ein neuer von der Modulschicht erstellt und eine Hierarchie aufgebaut, die auf übergeordnete Modulschichten sowie Klassenlader verweist. Dementsprechend referenziert der Modulschicht Klassenlader immer

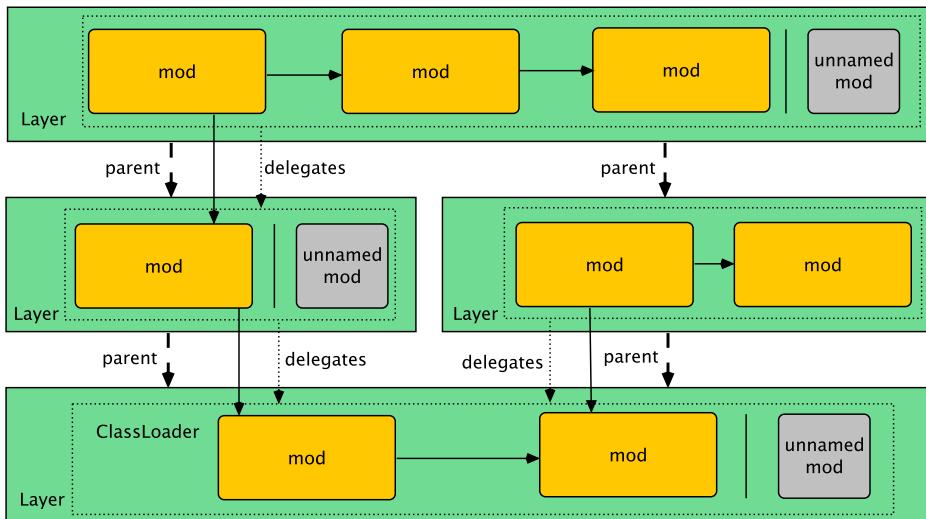


Abbildung 3.9: Modul Layer [?]

auf einen übergeordneten Klassenlader, um dem ehemaligen Delegierungsmodell zu entsprechen und die Funktionalität der *unbenannten Module* weiterhin erfüllen zu können. Des Weiteren referenziert die Modulschicht selbst auf eine variable Menge von übergeordneten Modulschichten, die das Auflösen der benötigten benannten Modulabhängigkeiten unterstützen und die Modulsuche an entsprechende Modulschichten weiterleitet.

Das neue Konzept der Modulschichten erweitert das bestehende System der selbst erstellten hierarchischen Namensräume, indem Modulabhängigkeiten beim Laden in die Applikation validiert, gruppiert und aktiv verwaltete werden können. Darüber hinaus enthält die Modulschicht Konfiguration die komplette Information über den Inhalt ihrer Schicht sowie der darunter liegenden Modulschichten.

Die Konfiguration von dem dynamisch nachgeladenen Code, der Schichten übergreifend validiert wird, verhindert das Überlagern der bereits integrierten Code und ändert die Art und Weise wie die Suche nach Modulklassen durchgeführt wird.

Die Klassensuche wird nun mit der *direkten Delegation* durchgeführt, die zuerst die eigene Schicht bevorzugt bevor die Anfrage an die darüber liegende Schicht weiterleitet wird. Die übergeordnete Schicht verhält sich nach demselben Prinzip und versucht zuerst selbst die Klasse zu laden bevor die nächste Delegation durchgeführt wird.

Kurzgefasst gehören Schichten zu Modulen wie Klassenlader zu Klassen. Ein Mechanismus zum Laden und Instanziieren von Elementen mit erweiterten Services und Sicherheitsbestimmungen.[?, ?, ?]

3.9 JDK Modulsystem

Um die Aufgestellten Regeln und Konzepte des entworfenen Modulsystems in Java zu integrieren, muss die Java Plattform diese selbst als Vorreiter erfüllen.

Demnach wurde die Laufzeitumgebung (*Run-Time*), die aus der *rt.jar* besteht, auf Module aufgeteilt und mit einander über Kommunikationsregeln und Schnittstellen verknüpft. Das Ergebnis der modularisierten *rt.jar* ergab 70 Module, die sich gegenseitig ergänzen.

In der Konsequenz ist eine kleine Applikation, die sich nur an dem *base* Modul bedient, mit einem Speicherbedarf von 16 MB umsetzbar. Im Gegensatz dazu müssten für ein paar Zeilen Code in der 8 Version von Java, 160 MB der *rt.jar* in die Laufzeitumgebung miteinbezogen werden, um den Laufzeitanforderungen zu entsprechenden.

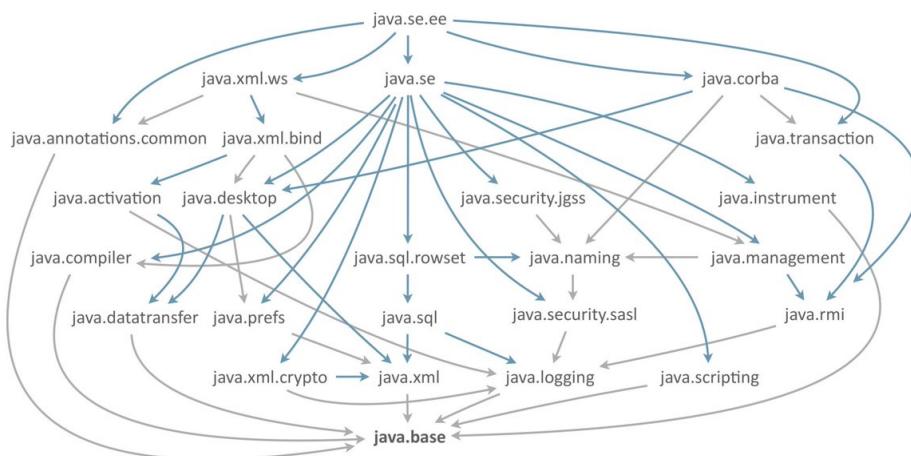


Abbildung 3.10: Modularisierte *rt.jar* Laufzeitumgebung [?]

Obwohl das Modulsystem viele Neuerungen und Aufwertungen der Java Plattform mit sich brachte, sind nicht alle Wünsche erfüllt worden. Wie zum Beispiel das Nutzen gleichnamiger Bibliotheken mit unterschiedlicher Versionsnummer auf demselben Modulpfad.

Nichtsdestotrotz gibt es Ansätze, die es erlauben eine Versionsnummer in einem Modul zu verankern und somit existiert die Wahrscheinlichkeit, dass dieser Fähigkeit in der Zukunft nachgerüstet wird.

Kapitel 4

Migration

In dem vorherigen Kapitel wurden Module und ihre Eigenschaften, Konstruktionsregeln sowie Modulararten behandelt. Dieses Kapitel beschäftigt sich mit der Fragestellung, wie Altsysteme, die vor dem Modulsystem von Java entwickelt wurden, in dem Modulkontext betrieben werden können und was getan werden muss, um diese den modernen Anforderungen anzupassen und zu modularisieren.

Die Migration behandelt Systeme, die nicht beständig auf dem aktuellen Stand der Technik gehalten werden können und an ihren Ausführungskontext gebunden sind. Diese rutschten langsam in den Bereich der Altsysteme, bis ihr Lebenszyklus zu Ende geht und der Legacy-Zustand erreicht ist. Um die Systeme weiterhin zu nutzen, müssen sie in eine Umgebung mit den geforderten Eigenschaften, ohne Änderungen der internen Struktur vorzunehmen, übertragen werden. Dieses Verfahren wird oft im Bereich der Softwaretechnik mit Software Reengineering und Software Neuimplementation verwechselt, dessen Ziel in der Optimierung der Codebasis liegt und nichts mit dem Ausführungskontext zu tun hat. [?]

Während des Betriebs einer lang gepflegten Kernapplikation, wird der Lebenszyklus öfter überschritten und muss den Migrationszyklus mehrmals durchlaufen. Zum Beispiel kann eine Applikation an Größe gewinnen und muss in die Cloud ausgelagert werden, die Anforderungen können sich verschieben und der Technologie-Stack muss an die Marktbedürfnisse angepasst werden, darüber hinaus kann der Ausführungskontext einen großen Versionssprung hinter sich lassen, der das Warten der Software unter den momentanen Bedingungen unmöglich macht.

Dem zufolge ist das Umfeld der Software Entwicklung eine dynamische Umgebung, denn auch mit einer gut durchdachten Architektur kann nicht garantiert werden, dass in der Zukunft heutige Paradigmen, Werkzeuge und Aufgabenbereiche denselben Kurs behalten. Deswegen existieren bereits sämt-

liche Migrationsstrategien, die als ein Leitfaden den Entwickler durch die Migration führen.

Im folgenden Kapitel werden Ansätze vorgestellt, die Monolithische- sowie Bibliotheksanwendungen in das modulare System überführen ohne Änderung an der internen Funktionalität durchzuführen.

4.1 Legacy-System

Der Begriff *Legacy-System* beschreibt ein altes System, das innerhalb einer Organisation länger als der geplante Lebenszyklus in Betrieb bleibt. Der englische Begriff *Legacy*, zu Deutsch Erbe, bezieht sich nicht auf das Alter der Software, sondern auf die Interpretation der Software als Erbe. Da die Umsetzung von früheren Entwickler Teams durchgeführt wurde, die sich an damaligen Konzepten und Strategien bedienten, ergab die Lösung ein Erbe mit bestimmten Einschränkungen, die für die zukünftige Erweiterung der Software eine große Rolle spielen. Denn, die typischerweise veralteten Verfahren und Technologie lange Lebenszyklen mit umfangreichen Veränderungen und Erweiterungen erfahren haben [?].

Zu meist handelt es sich um sogenannte Kernsysteme zur Unterstützung wesentlicher Geschäftsprozesse eines Unternehmens. Sie sind in der Regel geschäftskritisch und können nicht ohne großen Aufwand und Risiko für das Unternehmen ausgetauscht werden. Aufgrund ihres langen Lebenszyklus, ihrer Komplexität und ständigen Überarbeitungen ist die Logik solcher Systeme oft unübersichtlich und schlecht dokumentiert. Ihre Implementierung kann zusätzlich früher geltenden Standards unterliegen und anderen Programmierparadigmen folgen, die nur schwer verständlich sind. Daher sind Geschäftsprozesse und Geschäftsregeln im Code versteckt und müssten z. B. für eine Neuimplementation erst rekonstruiert werden [?].

4.2 Migration

Die Softwaremigration bezeichnet die Überführung eines Softwaresystems in eine andere Zielumgebung oder in eine sonstige andere Form, wobei die fachliche Funktionalität unverändert bleibt. Als Ausgangspunkt steht dabei immer ein bestehendes System, das an Anforderungen und Techniken des Anwendungsbereiches angepasst werden muss [?]. Die Adaption an den neuen Anwendungsbereich geschieht zu meist nicht problemlos und muss system- und kontextbezogene Hürden überwinden.

4.3 Migrationshürden

Die Migrationshürden sind fest mit dem Anwendungskontext verbunden und hängen stark von der Beschaffenheit der neuen Umgebung ab. Da in unserem Fall die Migration innerhalb der Java Umgebung stattfindet, müssen die Neuerungen des Modulsystems analysiert und auf den bestehenden Zustand der Applikation abgebildet werden.

- Die Probleme des Modulsystems beginnen mit den Zugriffsrechten auf die Core-JDK API's. Diese sind ab sofort in dem Modul gekapselt und bieten nur eingeschränkte Möglichkeiten Aufrufmöglichkeiten. Nichtsdestotrotz stellt Java für viele der gekapselten API's einen Ersatz zur Verfügung, wodurch zahlreiche Probleme mit einem relativ geringen Aufwand behoben werden können. [?, ?, ?, ?]
- Im Weiteren verbietet das neue Modulsystem namensgleiche Pakete in verschiedenen Modulen. Dieses adressiert das vorher besprochene Problem aus dem Kapitel ??, nämlich den Zugriff auf privat deklarierte Pakete aus fremden Modulen. Trotz dem gibt es Bibliotheken mit ähnlicher Paketstruktur, die sich nicht böswillig sich Zugriff verschaffen wollen, sondern spiegeln eine Standardstruktur einer Bibliothek widerspiegeln, wie zum Beispiel *de.firma.input.reader* kann in mehreren Bibliotheken eines Unternehmens existieren kann und ab Java 9 nicht mehr zulässig sein wird. Somit müssen Module mit ähnlicher Struktur angepasst werden, um den nächsten Modularisierungsschritt durchführen zu können. [?, ?, ?, ?]
- Der Klassenlader-Typ des Applikation-Klassenladers wurde überarbeitet und infolgedessen auch das Arbeiten mit den ehemaligen URL-Klassenlader Methoden. Der bestehende Code, der den URL-Klassenlader exzessiv nutzt und zum Beispiel Ressourcen aus verschiedenen Quellen lädt, muss auf den *Secure-Klassenlader* oder *Klassenlader* aufgewertet werden, um funktionstüchtig zu bleiben. [?, ?]
- Einer der kritischen Veränderungen, die die Modultatform mit sich bringt, ist das Verbot von zyklischen Abhängigkeiten von Modulen untereinander. Diese dürfen sich nicht gegenseitig mit den Schlüssel *require* koppeln, da sonst eine Veränderung in einem Modul zwangsläufig eine Änderung in den anderen Modulen hervorrufen kann. Dieser Stil kann sich schnell über die ganze Applikation verbreiten und kleine Änderungen an einer Stelle zu unübersichtlichen Seiteneffekten führen. Genau diese Probleme adressiert das Modulsystem in erster Linie und verbietet aus diesem Grund Zyklen in dem Applikationsentwurf. Um bestehende Zyklen in einer Applikation zu lösen, muss die Funktionalität genau betrachtet und in kleine unabhängige Aufgaben aufgeteilt

werden. Somit werden Zyklen aufgebrochen und die Aufgabenstellung jedes Moduls klar definiert. [?, ?, ?]

4.4 Migrationsarten

Da jede Applikation spezifisch Migrationsanforderungen besitzt, gibt es unterschiedliche Verfahren, die sich bestimmten Kriterien widmen. Dementsprechend sollte man die gegebene Applikationsbeschaffenheit ermitteln und dessen Probleme auf die passende Migrationsstrategie abbilden. Die prominenten Migrationsstrategien der Software Techniken sind *Chicken Little* und *Butterfly*, zwei der gängigsten Arten der Softwaresystem Migration. [?]

Softwaresysteme, die nach dem *Chicken-Little-Ansatz* migriert werden, zerlegen das System in mehrere Migrationspakete, die einzeln in kleinen inkrementellen Schritten in die neue Zielumgebung überführt werden. Damit der Betrieb des Systems nicht für die Zeitdauer der Migration ausfällt, existieren alte, neue und migrierte Teilsysteme nebeneinander. Die korrekte Kommunikation der Softwarebausteine muss über entsprechende Kommunikationskanäle errichtet werden und gestaltetet damit eine Brücke zwischen Alt- und Neuentwicklung, die zusammen eine gemeinsame Ressourcenbasis nutzen können. [?]

Der *Butterfly-Ansatz* geht von einer separaten Entwicklung der Applikation in der neuen Umgebung aus. Dies hat zur Folge, dass die Altapplikation in Betrieb bleibt und unverändert ihre Aufgabe erfüllt, bis der Nachfolger, der parallel zu dem Betrieb entwickelt und getestet wird, die Funktion korrekt widerspiegeln kann. Im Anschluss werden Ressourcenbestände in kleinen Paketen an die Applikation im neuen Kontext transferiert und das Altsystem abgeschaltet. Das *Butterfly*-Verfahren vermeidet also während der Migration den simultanen Zugriff auf Legacy- und Zielsystem. [?]

Zwischen den beiden vorgestellten *Migrationsideen* hat sich das Java Team für die Unterstützung der Schrittweise-Migration entscheiden, die eine Applikation in kleine Softwareeinheiten aufteilt und langsam auf den neuen Modulpfad migriert. Der parallele Betrieb der neuen sowie alten Struktur wird durch eine interne Brücke, die korrekte Kommunikation zwischen den Klassenpfad und den Modulpfad errichtet umgesetzt. Die Implementation der Kommunikationsbrücke und ihrer Charakteristika, die eine kritische und verantwortungsvolle Aufgabe in dem Migrationsprozess trägt, wird für uns von dem Java Team zur Verfügung gestellt. Somit unterstützt und führt das Modulsystem von Java den Entwickler bei der Hand zu der korrekten, sicheren, modernen und funktionstüchtigen Modulararchitektur.

Ungeachtet dessen, ist die Migration einer Applikation zu diesem Zeitpunkt

nicht zwingend und kann innerhalb des Modulsystems weiterhin betrieben werden, da diese keine spezielle Hilfsmittel von der Umgebung fordern.

In den nachfolgenden Kapiteln werden unterschiedliche Varianten der Migration und des Betriebs einer Applikation innerhalb des Modulsystems vorgestellt.

4.4.1 Plattform Migration

Die simpelste Migration ist eine reine Plattform Migration, ohne Änderungen an der Software vorzunehmen, wie in der Abbildung ?? dargestellt. Dies ist möglich, dank der Unterstützung des Klassenpfades, der in diesem Fall unsere Applikation als ein *unbenanntes Modul* im Modulsystem betreibt. Wie bereits im Kapitel ?? angesprochen, bietet dieses Modul den Betrieb der Legacy Anwendung in der neuen Umgebung mit dem geringen Anteil der möglichen Vorteile, wie zum Beispiel den Sicherheitsupdates. Zusätzlich behält die Applikation ihre monolithische Architektur und vermisst alle Modulsystem Features, die im Kapitel ?? und ?? angesprochen wurden.

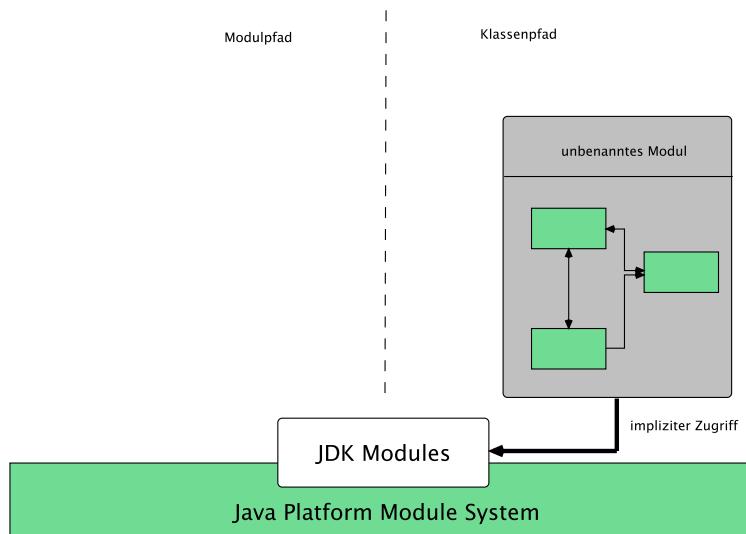


Abbildung 4.1: Plattform Migration [?]

Eine weitere Möglichkeit wäre es, die Applikation und ihre Bibliotheken auf den Modulpfad zu migrieren und als *automatische Module* zu betreiben. So mit wäre der alte Klassenpfad nicht mehr im Applikationsdesign vorgesehen und fokussiert die Entwickler auf die Arbeit mit dem Modulpfad. Dennoch lässt sich nicht jede Anwendung in diesem Modus migrieren, denn die Hürden aus den Kapitel ??, wie die *Zyklen-Freiheit* sowie der verbot der *Split-Packages* in Legacy Bibliotheken nicht immer gegeben ist.

4.4.2 Top Down Migration

Die *Top-Down* Migration behandelt die Migration von oben nach unten, dabei werden zuerst die Applikation und im Nachhinein die Drittanbieter Bibliotheken migriert. Dafür müssen die Applikationspakete auf Abhängigkeiten geprüft und entsprechende Module sowie Modulbeschreibungen nach den im Kapitel ?? besprochenen Kriterien erstellt werden. Mit dieser Methode werden die Drittanbieter Bibliotheken zuletzt betrachtet, da es sich bei diesen um Fremdcode handelt.

Da nach der Prüfung der Abhängigkeiten ein Graph entsteht, verweist dieser ab einem gewissen Punkt auf die Bibliotheksabhängigkeit der Applikation, die wiederum von weiteren Bibliotheken abhängen. Somit kann die Wurzel der benötigten Drittanbieter Bibliotheken einer Applikation ausfindig gemacht und als *automatische Module* in den Modulpfad eingebunden werden. Somit können diese, wie bereits in im Kapitel ?? behandelt, sowohl mit der Applikation als auch mit den Legacy Elementen ihrer eigenen Abhängigkeiten zugleich interagieren. Zum Schluss kümmert man sich um die Bibliotheken auf dem Klassenpfad, indem diese ersetzt oder selbstständig weiterentwickelt werden, mit dem Ziel die Gesamtapplikation auf dem Modulpfad zu übertragen. [?, ?, ?, ?, ?]

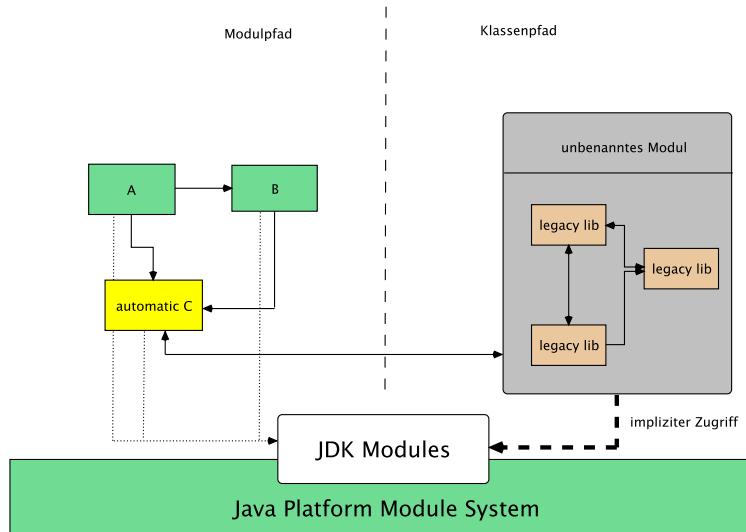


Abbildung 4.2: *Top Down* Migration

Diese Migration ist besonders vorteilhaft bei Applikationen, die eine geringe Codebasis besitzen und in einem kurzen Zeitraum eine modularisierte Form annehmen können.

4.4.3 Bottom Up Migration

Die *Bottom Up* Migration behandelt lose Module zuerst, denn diese haben keine Abhängigkeiten und bieten eine gekapselte Funktionalität an. Um herauszufinden, welche Module sich für den initialen Migrationsschritt eignen, kann der Abhängigkeitsgraph mithilfe des *JDepth* erstellt werden. Module, die als Blätter aufzufinden sind, können zuerst migriert werden, da sie keine Kinderknoten und somit keine Abhängigkeiten besitzen. Im weiteren Verlauf der Migration werden die übrig gebliebenen und neu entstandenen Blätter des Abhängigkeitsbaums abgearbeitet, bis die ganze Applikation, samt der Drittanbieter Bibliotheken, sich auf dem Modulpfad befindet. Während der Migration müssen natürlich die im Kapitel ?? besprochenen Kriterien erfüllt werden, um eine robuste Umsetzung zu erreichen. [?, ?, ?, ?, ?, ?]

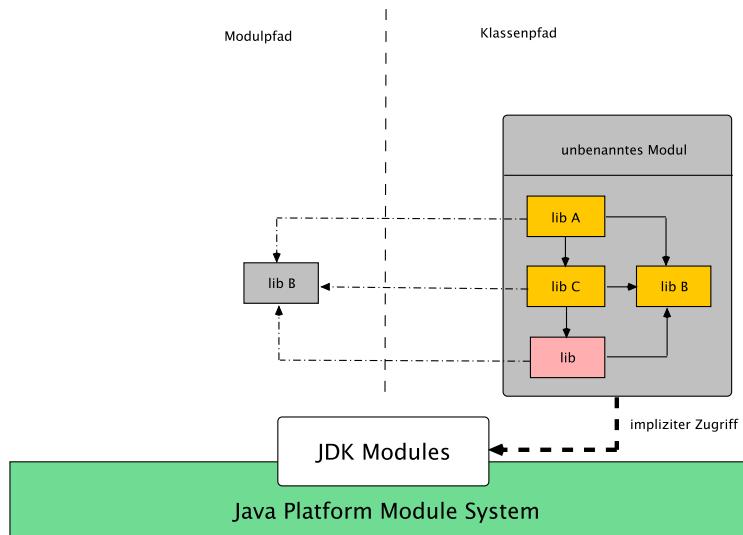


Abbildung 4.3: *Bottom Up* Migration [?]

In der Abbildung ?? wird der erste Schritt der *Bottom Up* Migration ange deutet, indem die *lib B* als erste auf den Modulpfad migriert wird. Diese hat keine Abhängigkeiten und ist der perfekte Kandidat für den initialen Schritt. Als Nächstes bietet sich die *lib* Bibliothek für die Migration an, da sie keine weiteren Abhängigkeiten in den Klassenpfad besitzt. Jedoch ist sie eine Drittanbieter Bibliothek und kann von uns nicht bearbeitet werden. Aufgrund dessen wird die *lib C* für den nächsten Migrationsschritt ausgewählt und als ein *automatisches Modul* auf den Modulpfad migriert. Somit kann dieses die *lib* und *lib B* zugleich nutzen. Als Nächstes ist die *lib A* an der Reihe, dessen komplett Abhängigkeiten sich bereits auf dem Modulpfad befinden. In der Konsequenz befinden sich alle Applikationsbibliotheken auf dem Modulpfad. [?, ?, ?, ?, ?, ?]

Die *Bottom Up* Migration bietet sich bei bereits aufgeteilten Applikationsarchitektur an, die über eine Menge von *Jar* Bibliotheken betrieben wird. Diese braucht weniger Aufwand, um den Monolithen zu zerlegen und sind bereits über Schnittstellen mit einander verbunden. [?, ?]

Kapitel 5

Analyse der Ausgangssituation

Dieses Kapitel diskutiert die Motivation für die Abschlussarbeit, setzt klare Ziele sowie Rahmenbedingungen und erörtert den Einfluss des Java Modulsystems auf RENEW wie auch die derzeitige Plugin-Architektur. Des Weiteren wird ein Zustand ermittelt, der als Basis für die nachfolgenden Prototypen gelten wird.

5.1 Motivation

RENEW ist ein Petrinetz Simulator, der von dem Arbeitsbereichs TGI entwickelt wird und unterstützt das Erforschen von komplexen Systemen auf der Grundlage formaler Modelle.

Weil RENEW in Java geschrieben ist, ist die Applikation an die Java Plattform angewiesen und muss dementsprechend den Plattform Anforderungen und Richtlinien folgen. Die Anforderungen und Richtlinien einer Plattform sind nicht fix und können sich ändern, besonders mit einem großen Versionssprung. Das neu eingeführte Modulsystem von Java, wird mit einem großer Versionssprung in Verbindern gebracht und setzt neue Normen und Anforderungen für das Entwicklung von Java Applikationen.

Es gibt mehrere Gründe warum RENEW die Migration auf das Modulsystem durchführen sollte. Im Folgenden werden die wesentlichen Argumente, die für die Migration sprechen diskutiert.

5.1.1 Architektur

Die initiale Entwicklung von RENEW begann mit einer monolithischen Architektur. Diese erfüllte die nötigen Anforderungen, eignet sich jedoch nicht für Entwickler mit geringer Kenntnis über die Gesamtarchitektur und den darunterliegenden theoretischen Konzepten. Daher wurde eine Plugin-Architektur

aufgesetzt, die es ermöglichte Studenten RENEW mit Logik innerhalb eines Plugins zu erweitern. Dieses Verfahren trägt bereits den Gedanken der Modularisierung in sich, da die Gesamtarchitektur in Bestandteile zerlegt und mit einander entkoppelt verknüpft wird. Somit wäre die Einführung des Modulsystems von Java der nächste Schritt in Richtung erweiterbare und zusammensetzbare Systeme.

5.1.2 Verkürzter Entwicklungszyklus

Die Aufteilung einer monolithischen Architektur auf eine Plugin-Architektur war ein großes Ereignis für Renew. Denn mit der Zerlegung der Gesamtarchitektur, wurde die Komplexität auf die entstandenen Komponenten aufgeteilt und erlaubte eine mühelose Weiterentwicklung der Applikation über die Plugins.

Obwohl die RENEW Plugin-Architektur lange im Betrieb blieb, hatte das Plugin-System die Codebasis umorganisiert, ohne diese zu verändern. Diese führt zu altem, unverständlichem Code aus der Java 1.4 (2002) Version, mit dem viele Konzepte und Architektur Entscheidungen getroffen wurden. Nach fast 18 Jahren Betrieb altert die Codebasis, die Ideen und Konzepte für die Umsetzung ihrer Funktionalität. Besonders konfus und aufgebläht können Funktionsumsetzungen erscheinen, die heute von Java 12 in ein paar Zeilen gelöst werden können. Der zügige und rapide Wandel der Software Paradigmen und deren optimaler Einsatz in der Software Architektur ist ein Teil des Fortschritts und muss in die Planung des Lebenszyklus der Applikation mit einkalkuliert werden.

Dementsprechend ist die Modularisierung und dessen Anforderung an die Struktur und Inhalt ein wichtiges Ereignis für den RENEW Lebenszyklus. Denn dieser erreicht wieder sein Ende und wird mit dem Modularisierungsschritt zurückgesetzt.

RENEW's Entwicklungseinheit ist das Plugin. Diese repräsentiert ein bestimmtes Feature mit einem eigenen Lebenszyklus, wie zum Beispiel ein Formalismus, Simulator oder Fenster Management Plugin. Diese müssen Daten entgegennehmen, diese verarbeiten und wieder ausgeben. Demzufolge bündelt ein Plugin mehrere Fähigkeiten, die zusammen ein Feature verkörpern. Der wesentliche Nachteil einer Codeänderungen in einem größeren Plugin, ist das Beeinträchtigen des Gesamtverhaltens des Plugins und fordert dementsprechend ein komplettes Testszenario aller Plugin Fähigkeiten.

Mit der Einführung der Module, kann das Plugin in kleinere Einheiten zerlegt werden, die anschließend eine gekapselte Teilfunktionalität des Plugins in sich tragen und Auswirkungen der Modifikation eingrenzen. Diese sind klein, leicht änderbar, ersetzbar und besitzen einen eigenen unabhängigen Lebenszyklus. Somit verkürzt sich die Entwicklungsdauer einer Änderung

innerhalb eines Plugins, da der Einfluss auf die Umgebung durch die Modulgeneren eingeschränkt ist. Des Weiteren bieten Module eine Möglichkeit kooperativ und parallel an einem Plugin zu arbeiten, indem die Module gegen vereinbarte Schnittstellenbeschreibungen entwickelt werden.

Demnach erweitert die Modularisierung den RENEW Kontext und erlaubt das Entwickeln von Plugins in Rahmen eines Studenten Projekts, indem Teilaufgaben eines Plugins auf Module zerlegt und parallel von Studenten bearbeitet werden können. Darüber hinaus ist das Zusammenführen der Ergebnisse eine konfliktfreie Angelegenheit und bedarf keiner kompletten Gruppenaufmerksamkeit, um die passenden Codeblöcke für die Gesamtfunktionalität auszuwählen, da es sich so gut wie keine Überschneidung in der Aufgaben Implementation bilden kann. Somit profitiert RENEW von den kurzen Entwicklungszyklen der Module und deren unproblematischen Verknüpfungseigenschaften.

5.1.3 Code-Bausteine

Eine der wichtigsten Fähigkeiten eines Entwicklers, ist die Beherrschung der Komplexität. Diese führt zu sauberem, lesbarem, wartbarem Code und erweitert den Lebenszyklus einer Software um ein Vielfaches. Um diese Kompetenz zu meistern, bietet das Modulsystem von Java unterstützende Werkzeuge, die den erstellten Code organisieren und strukturieren, um ein langlebiges Ergebnis zu erzielen.

Da RENEW das Produkt vieler Abschluss-, Projekt- und Promotionsarbeiten ist, durch die die Software ihre Gestalt annimmt, gibt es diverse Beschäftigte mit eigenen Zielen und Interessen. Daher ist eine allgegenwärtige, globale Strukturanforderung, die jedem Entwickler bekannt ist und die verpflichtend eingehalten werden muss, eine erstrebenswerte Charakteristik.

Die im Kapitel ?? vorgestellten Modul Charakteristiken beschreiben die von dem Java Modulsystem eingesetzten Richtlinien für die saubere Softwareentwicklung und erzwingen zum Teil ein Still der fein granulierten Code-Bestandteile, die kombiniert ein Softwaresystem ergeben.

Die Charakteristiken fördern den Entwickler zum Entwickeln von abgeschlossenen Einheiten auf und verhindern somit das Entstehen von den sogenannten *Spaghetti Code*, der funktionsübergreifende Anpassungen trifft und den Überblick über den Zusammenhang der Gesamtarchitektur unscharf erscheinen lässt.

Die Module und die entsprechenden Richtlinien erschweren den *Spaghetti Code*, indem Mehraufwand für die Kommunikation zwischen den Modulen

erbracht werden muss und machen das unsaubere Arbeiten unattraktiv. So mit dienen Module als Grenzen für den Entwicklungsrahmen eines Features und engen den Bearbeitungs- und Betrachtungsraum für den Entwickler ein. Daraus ergibt sich ein Softwarepaket, das unabhängig von den Senior-Entwicklern verstanden, genutzt und angepasst werden kann, da der Aufbau nicht mehr in dem Wiki, Readme oder beim Entwickler selbst verankert, sondern direkt in der Codebasis integriert ist.

Demzufolge profitiert RENEW von der Modularisierung, indem sich immerfort wechselnden Akteure eine saubere Codebasis hinterlassen, die den nächsten Absolventen sowie den wissenschaftlichen Mitarbeitern viel Zeit erspart.

Aus einer sauberen Umsetzung folgen saubere Code-Bausteine, die wieder verwendet werden können. Die genannten Eigenschaften der Module bringen einen wesentlichen Vorteil beim Optimieren der RENEW Applikation, indem kontextbezogen Module ausgetauscht werden können, um ein besseres, lokales Ergebnis zu erzielen. Zum Beispiel können zielgerichtet ausgewählte Plugins für die Erfüllung einer speziellen Aufgabe, wie das Validieren von P/T-Netzen, ein besseres Ergebnis abliefern, indem ein für diesen Anwendungsfall angepasste Verarbeitungsalgorithmus angewandt wird. Dieser ist natürlich in einem Modul gekapselt und besitzt Schnittstein identisch zu seinem Vorgänger. Auf diese Weise kann eine große Anzahl an Modulen mit gleicher Funktion und unterschiedlicher Zielsetzung erstellt werden, die in einem Modulkatalog verwaltet und bei Bedarf ausgetauscht werden können.

5.1.4 Code Management

Das Plugin Management ist ein zentrales Themengebiet von RENEW, da Plugins geladen instanziert und genutzt werden müssen. Diese wichtige Aufgabe übernimmt der *PluginManger* innerhalb des *Loader* Plugins, der Plugins in den Klassenpfad lädt und mit den benötigten Bibliotheken ausstattet. Das Ladend geschieht über den Plugin Klassenlader, der den Code aus den gegebenen Quellen auf den Klassenpfad platziert.

Der große Nachteil des Klassenpfad, ist die Auflösung von Archiv Grenzen und somit auch von Plugin Grenzen. Das heißt, Java kann nicht unterscheiden aus welcher Bibliothek die Klasse stammt und muss aus diesem Grund den kompletten Klassenpfad für eine Plugin Klassenabfrage durchsuchen. Infolgedessen ist das Überwachen der Plugins auf dem Klassenpfad nur schwer möglich, da der Klassenpfad nur Informationen über die Klassen und ihre Pakete besitzt **??**. Des Weiteren ist die Zuordnung einer Klasse zu ihrer Quelle ebenso Problematisch.

Dieses Verfahren wurde ständig bemängelt und wird mit der Einführung des Modulsystems von Java adressiert, indem die Konfigurationsdateien *module-info.java* eingeführt wurde, die der Klassen Quelle eine Identität verleiht.

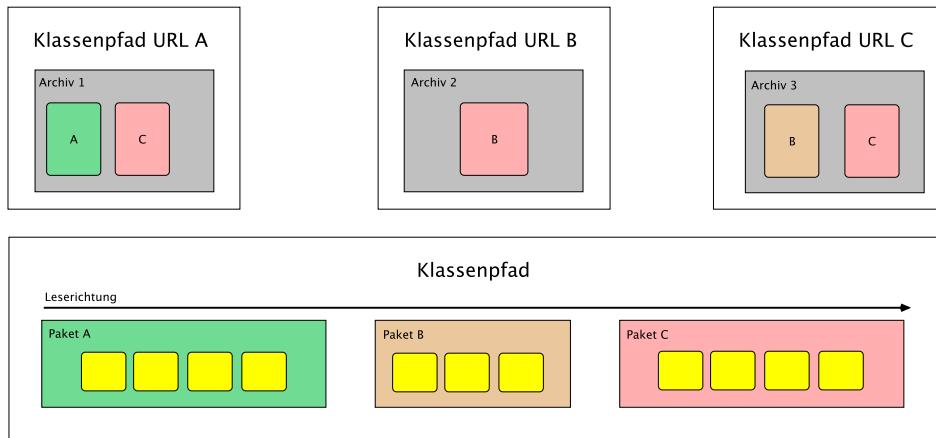


Abbildung 5.1: Klassenpfad Suche

Demzufolge ist ein Archiv oder Verzeichnis nicht mehr ein aussagen loser Behälter, sondern ein Objekt, das Information über seine Kennung, seinen Inhalt und seiner Kommunikationspartner besitzt.

Dies bringt zwei wesentliche Vorteile für die Arbeit mit dem Plugin Code. Die erste Eigenschaft beschreibt Referenzen auf Archiv-Objekte, die die benötigten Klassen besitzen. Der Klassen werden nicht mehr von links nach rechts durchsucht ??, wie es vorher der Fall war, sondern über das Modul mit den entsprechenden Klassen adressiert ??.

Das bewusste Nachschlagen nach Klassen erhöht die Performance und setzt zusätzlich das eindeutige und verantwortliche Modul für die gegebene Implementation fest. Darüber hinaus erleichtert die eindeutige Modulzuordnung das Verfolgen von unerwarteten Verhalten, wie zum Beispiel das Überdecken von Klassen von weiteren Klassen desselben Typs, die sich weiter vorne im Klassenpfad befinden.

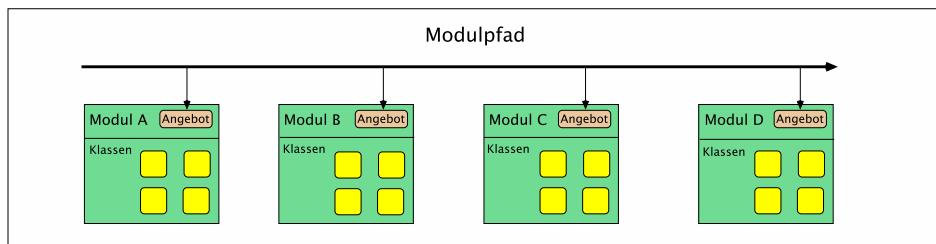


Abbildung 5.2: Modulpfad Suche

Der zweite Vorteil, beschreibt das Auslesen der Modulinformation aller Module, die sich auf den Modulpfad befinden. Demzufolge können Plugins in Archive verpackt und auf dem Modulpfad wiedergefunden, analysiert und

sogar bearbeitet werden. Somit bring Java ein neues Instrument zum Verwalten der Codebasis von modularisierten Anwendungen, die dem Nutzer bei Bedarf die internen Bausteine präsentieren lässt.

RENEW kann mithilfe der neuen API des Modulsystems die Verwaltung der Plugins erweitern, indem geladenen Plugins direkt angesprochen und manipuliert werden. Zum Beispiel könnten ausgewählte Plugins nur für bestimmte Nutzer oder Modi ihre Funktionalität anbieten oder eine angenehme Visualisierung der laufenden Plugins und dazugehörige Funktionalitäten dem Nutzer präsentieren. Des Weiteren ist die performante Suche nach Klassen in einem dynamischen System wie RENEW, eine erstrebenswerte Qualität.

Im folgenden Kapitel der Auswirkungen ?? wird auf die *Modulschicht* eingegangen, die das zielorientierte Laden nur notwendiger Module und somit nur notwendiger Plugins ermöglicht.

5.1.5 Laufzeit-Abbildung

Für die Nutzung von Java Applikationen ist immer ein installierte Laufzeitumgebung notwendig, die Versionskompatibel mit der entsprechenden Applikation sein muss. Dementsprechend muss Java installiert und eingerichtet werden bevor unser Code ausgeführt werden kann. Für große Serveranwendungen ist der Aufwand gerechtfertigt, da sie unikale, langlebige und komplexe Gebilde darstellen. Für Anwendungen die kleiner sind und dynamisch zwischen Hardwareknoten verteilt werden müssen, ist das manuelle Anlegen der Laufzeitumgebung eine mühselige und fehleranfällige Aufgabe.

Das Modulsystem von Java hat diese Herausforderung mithilfe der *Laufzeit-Abbildung* adressiert. Die *Laufzeit-Abbildung* besteht aus einer Verzeichnisstruktur, die alle notwendigen Komponenten für den Betrieb der Applikation besitzt. Dazu zählen Module, Skripte, native Bibliotheken, Konfigurationen, Dokumentationen sowie Lizenzbedingungen und sogar Sicherheitsmechanismen zum Validieren der Komponenten sind vorgesehen. Die generierten *Laufzeit-Abbildungen* sind in sich Abgeschlossen und können einfach auf beliebig viele Hardwareknoten automatisiert verteilt und ausgeführt werden, ohne zusätzliche Justierung der Software Umgebung.

Die Eigenschaft der Abgeschlossenheit der *Laufzeit-Abbildung*, kann von RENEW aufgegriffen und genutzt werden, um ausgewählte Plugin Gruppen horizontal nach Belieben zu skalieren. Zusätzlich hält die *Laufzeit-Abbildung* nur notwendige Java sowie Benutzer erstellte Module, die bestens für die Ausführung der Applikation auf einander abgestimmt werden. Infolgedessen kann das modularisierte RENEW auf optimale Ausführungsgeschwindigkeiten zählen.

5.1.6 Vision

Der zeitgemäße Zustand einer Applikation ist ein Zeichen hoher Qualität und reflektiert enorme Ansprüche an den Betrieb der Applikation. Diese kann geschäftskritische Qualitäten tragen, die den marktführenden Vorteil bringt und der Konkurrenz ein Schritt voraus ist. Um den Vorsprung zu sichern, ist eine vorausschauende Flexibilität gefragt. Mithilfe dessen die Applikation in der Lage ist, mit minimalem Aufwand, an die führenden Technologien anzuknüpfen.

Die aktuell führenden Trends beschäftigen sich mit der verteilten und wieder verwendbaren Softwareumsetzungen, die ständig an Komplexität gewinnen und trotzdem leicht beherrschbar bleiben muss. Diese beschreiben Ansätze wie gewisse Ziele erreicht werden können und setzen Grundvoraussetzungen zum Erreichen dieser Ziele. Dementsprechend muss RENEW bestimmte Grundvoraussetzungen erfüllen, um die Vorteile der Trends zu Nutzen und den Schritt mit dem Fortschritt zu halten.

Zum Beispiel wäre die Docker Umgebung für RENEW eine willkommene Erweiterung, mit der interne Bestandteile distributiv betrieben werden können. Somit wäre die Ausführung von RENEW nicht mehr an eine Maschine gebunden und kann bei Bedarf horizontal skaliert werden. Im Folgenden stellt sich die Frage: welche internen Strukturen von RENEW müssen individuell behandelt und anschließend kooperativ zusammengeführt werden. Auf diese Frage gibt es keine pauschale Antwort, jedoch ist es klar, dass die Plugins von RENEW feingranular betrachtet werden müssen, um sich ein Bild der Verarbeitungskette zu erstellen und diese den Bedürfnissen anzupassen.

RENEW auf verschiedenen Hardwareknoten zu verteilen ist nur der erste Schritt der distributiven Ausführung. Es fehlt die Koordination zwischen den Knoten, die die Verarbeitung koordinieren und die Ergebnisse zusammenfassen. Somit gibt es eine weitere Technologie, die sich dieser Aufgabenstellung widmet: Der Mikroservice Architekturansatz, der sich um die Koordination und das Zusammenspiel von Applikationsschwärmen kümmert.

Mithilfe der Mikroservicearchitektur und der Docker-Umgebung wäre die distributive Ausführung von RENEW erreichbar, doch zuerst muss RENEW den aktuellen Stand der Technologie entsprechen und demzufolge das Modulsystem von Java integrieren.

5.2 Folgen

Die Folgeerscheinung der Modularisierung unterbindet zahlreiche Entwicklungsschwächen, wie die *Code Organisation*, *Zyklen*, *Split Packages* und anti quierte API's. Diese sollen mit der Migration auf das Modulsystem von Java

aufgelöst, reorganisiert und nachgerüstet werden, um einen kompilierefähigen Zustand erreichen zu können.

5.2.1 Projektstruktur

Das Plugin System von RENEW besteht aus einer großen Anzahl an Plugins, die Zweckorientiert kombiniert werden können. Demnach besteht bereits ein Organisationsaufwand, der mit jeder Plugin Kombination wächst.

Zu den bestehenden Organisationsaufwand wird eine neue Ebene der Administration eingeführt, nämlich die Ebene der Module. Diese zerlegen Aufgabenmengen aus einer Code-Struktur in mehrere Code-Bausteine, die infolgedessen innerhalb eines Plugins disjunkt verwaltet werden müssen.

Da die Kombination aus Plugins und Modulen ausschlaggebend für die RENEW Funktionalität ist, muss eine passende Projektorganisation eingerichtet werden, die jedem Plugin die Möglichkeit bietet, zusätzliche Module mit dem Quell-Code, den Ressourcen sowie den Tests anzulegen.

Des Weiteren spielt die Paketorganisation eine zunehmend stärkere Rolle im Modulsystem von Java, denn eine Überschneidung eines Paketnamens mit anderen Bibliotheken im System ist ausgeschlossen und muss dementsprechend einen global eindeutigen Namen besitzen.

Die Einschränkung in der Namensgebung und Code Organisation ist für RENEW eine substantielle Änderung. Die Plugins wurden von unterschiedlichen Entwicklern zu unterschiedlichen Zeitpunkten entwickelt, ohne eine einheitliche Struktur Plugin übergreifend einzuhalten. Daraus folgt eine wilde Ressourcen- und Java Fremdcode Allokation, die das Lesen, Analysieren und bearbeiten der Plugins schwierig gestaltet. Zum Beispiel befinden sich gewisse Ressourcen im Plugin Wurzelverzeichnis unter *tools*, *samples*, *ontology* und andere direkt im Java Source-Code unter *src/**/*.(gif/jj/rnw/sns/vm)*. Der Aufbau führt zu der Frage, wie kompatibel ist das Ressourcen Layout mit dem Modulsystem von Java. Denn, Ressourcen werden im Modulsystem von Java genauso wie die Java Klassen innerhalb eines Modul Pakets gekapselt und verwaltet. Demzufolge kann das Paket *samples* nur einmal in einer Modulmenge existieren, da das Modul global den kompletten Namensraum für sich und seine Ressourcen alloziert.

Das Problem der Ressourcenorganisation, kann mithilfe des *META-INF* Verzeichnis gelöst werden, denn das *META-INF* Verzeichnis ist ein besonderes Verzeichnis, das nicht als ein Klassenpaket interpretiert wird und dementsprechend keinen ausführbaren Code enthält. Demzufolge können namensgleiche Ressourcen Verzeichnisstrukturen in unterschiedlichen Modulen denselben Namensraum besetzen und alle benötigten Ressourcen wie Konfigurationsdateien, Bilder und komplexe Darstellungsobjekte enthalten.

5.2.2 Split Packages

Im Kapitel ?? wurde die neue Klassensuche innerhalb des Modulsystems von Java thematisiert. Die Klassensuche ordnet jedes Klassenpaket eindeutig einem Modul zu, um den Klassenpfad durch eine Struktur zu erweitern, die sicher, schnell und eindeutig ist. Für RENEW ist die Voraussetzung der eindeutigen Paketbenennung noch nicht erfüllt. Denn, RENEW besteht aus Plugins, die sich gegenseitig sowohl ergänzen als auch erweitern und neigen dementsprechend zum ähnlichen oder äquivalentem Aufbau. Zum einen geschieht es bedacht, um zusammengehörige Klassen auf dem Klassenpfad näher zu bringen und unter einem Paketnamen zu vereinen, zum anderen geschieht es zufällig, da sich der Aufbau gut für die Zielsetzung eignet.

Die Lösung für den ähnlichen Aufbau ist die Anpassung der Paketstruktur, die eindeutig für jedes Plugin definiert wird. Zum Beispiel müssen Plugins aus dem gleichen Kontext, wie *Navigator* oder *NavigatorGit*, die denselben `de.renew.navigator.*` Namensraum beanspruchen disjunkt aufgebaut und benannt werden, um eine eindeutige Identität zu etablieren.

Im Gegensatz zum ähnlichen Aufbau, ist das Erweitern von Plugins, indem gleichnamige Paketstrukturen aufgebaut werden, ein Komplexes und tief führendes Problem. Wie bereits im Kapitel ?? besprochen, darf es keine Überschneidung der Namensräume geben. Das heißt, die Annahme, dass die benötigten Klassen aus anderen Plugins sich in demselben Paket auf dem Klassenpfad befinden ist nicht mehr zutreffend.

Für RENEW hat es eine ganz besondere Bedeutung, da der Code nicht nur kompiliert, sondern zuvor generiert werden muss. Das Generieren geschieht mit unterschiedlichen Techniken, die zum Beispiel mit `rnw`, `sns`, `jj`, `mv` Dateiformaten arbeiten und für bestimmte Anwendungsfälle unterschiedliche Java Klassen sowie Ressourcen generieren.

Der generierte Code und die generierten Ressourcen orientiert sich auf das Zielpaket und nutzt zum Teil keine *imports*, sondern direkte Klassen Zugriffe, da man sich sicher ist, dass die benötigten Klassen von anderen Plugins innerhalb ihres Pakets nachgeladen werden. Dementsprechend liegt das Problem nicht in der Java Klasse, Ressource oder dem Plugin, sondern in der Verarbeitungskette der Generatoren, die variable Java Klassen sowie dazu passende Ressourcen für die Kompilation bereitstellen.

Um die *Split Packages* aufzulösen, müssen die Verarbeitungsketten von vorne bis hinten analysiert und angepasst werden. Dazu zählt das Auffinden der benötigten Java Klassen innerhalb der modularisierten Plugin Menge, das Verweisen auf den vollwertigen Klassennamen sowie das Öffnen und Konsumieren der entsprechenden Pakete aus den notwendigen Plugins. Des Weiteren muss der Anwender der Ressourcen den vollen Zugriff auf alle in der

Ressource referenzierten Klassen besitzen, um dessen Inhalt vollständig darstellen zu können. In der Konsequenz ist das Konvertieren der *Split Packages* Technik auf eindeutige Modulstrukturen eine verwurzelte Problematik.

Das Einsetzen der *Split Packages*, ermöglichte den Entwickler eine einfache Art und Weise Code aus unterschiedlichen Plugins zusammenzuführen. Jedoch hat sich die Technik auf lange Sicht nicht bewährt und wird mit dem Modulsystem von Java verworfen. Da RENEW über einen langen Lebenszyklus verfügt und diese Technik von Beginn an weitgreifend einzusetzen, müssen diese Mängel behoben werden bevor die Applikation auf den Modulpfad Einsatzbereit ist.

5.2.3 Projekt Verwaltung

Die Herausforderung dynamische Modulkombination aus einer großen Modulmenge zu erstellen, spielt eine zentrale Rolle in einem Modulverband. Dementsprechend ist die Administration ein wichtiges Instrument für das Entwickeln von Modularerem Code.

Für die Verwaltung der Codebasis existieren bereits Werkzeuge, die eingesetzt werden können wie zum Beispiel *Ant*, *Maven* und *Gradle*. Da *Ant* in RENEW eingesetzt wird und *Gradle* im Gegensatz zu *Ant* weiterführende Konzepte mit sich bringt, werden im Folgenden Gründe für den Umstieg von *Ant* auf *Gradle* formuliert.

Ant

Zurzeit wird *Ant* für das Kompilieren und Erstellen der ausführbaren Plugins Archiven in RENEW benutzt. *Ant* ist das älteste der vorgestellten Werkzeuge und setzt die Grundlage für alle nachfolgenden Verwaltungsinstrumente. Dieses ist imperative und teilt dem System mit, was getan werden muss und wie die Arbeit verrichten werden soll. Mit anderen Worten, es wird eine Reihe von Aktionsanweisungen bereitgestellt, die das System in derselben Reihenfolge ausführt. Dementsprechend ist *Ant* höchst flexibel und erlaubt eine feingranulare Konfiguration jedes einzelnen Ausführungsschritts. Wie zum Beispiel das setzen der Ortsangaben, das Aufteilen sowie das Koordinieren der ausführbaren Code-Menge und natürlich die Kalibrierung des Compilers für die Übersetzung, die verpflichtend durchgeführt werden muss.

Obwohl die explizite Konfiguration viele Möglichkeiten bietet, ist das Schreiben und Konfigurieren an keinen Konventionen oder Strukturen gebunden. Infolgedessen führt die Entwicklung zu riesigen XML-Dateien, die nur schwer zu verstehen und zu warten sind. Des Weiteren ist *Ant* auf die Laufzeitumgebung angewiesen und erwartet bestimmte Bibliotheken und Applikationen auf der Ausführungsmaschine vorinstalliert, wie zum Beispiel *Ant* selbst, *Javacc* und andere Applikationen relevanten Software, die aus dem Klassenpfad erreichbar sein müssen.

Renew und Ant

Da RENEW mit dem *Ant* Werkzeuge verwaltet wird, existieren bereits Skripte mit allen notwendigen Schritten für das Erstellen der *Renew* Plugins. Die Skripte sind in drei Kategorien unterteilt, um die Projekt Verwaltung in RENEW zu etablieren. Zuerst wurden allgemeinen *task* und *target* Skripte eingeführt, die für alle Plugins benötigt werden, um die wiederholenden Konfigurationen zu minimieren. Des Weiteren wird für jedes Plugin ein eigenes Skript mit dem entsprechenden Lebenszyklus eingerichtet, der aus den allgemeinen *tasks*, *targets* und zusätzlichen Plugin bezogenen Aufgaben zusammengesetzt wird und zum Schluss ein ausführbares Ergebnis liefert. Die innovative Idee des Lebenszyklus ähnelt stark der *Maven* Umsetzung, die in der Zukunft mit einem integrierten *out of the box* Ansatz werben wird. Dennoch ist die derzeitige *Ant* Umsetzung der RENEW Umgebung verbos und wiederkehrend.

Nachdem alle Plugins Konstruktion fähig sind, wird ein globales XML-Skript eingerichtet, das alle Plugin Skripte referenziert und verwaltet. Zum Beispiel zählt dazu das Erstellen von Plugin Zusammenhängen und ihre Bearbeitungsreihenfolge.

Verwaltung der Drittanbieter-Bibliotheken

Obwohl *Ant* das Bauen von Java Applikationen automatisiert, war die Verwaltung der Projektabhängigkeiten nicht ein Teil der Umsetzung. Dementsprechend müssen direkte sowie transitive Projektabhängigkeiten von den Entwicklern mitgeliefert und in die entsprechenden Klassenpfade der ausgewählten Projekt- *task* und *targets* konfliktfrei eingebunden werden.

Die Verwaltung der benötigten Bibliotheken ist mühselig und bedarf einen enormen Verwaltungsaufwand, um alle transitiven Abhängigkeiten aufzulösen und Konflikte zu beseitigen. Die Anforderung der höchst gewünschten Abhängigkeitsverwaltung wurde erst in der nächsten Generation von *Maven* umgesetzt und im Anschluss von einem Tochterprojekt von *Ant* namens *Ivy* übernommen.

Momentan verwaltet RENEW die Drittanbieter Bibliotheken manuell und kann die Organisation der Abhängigkeiten bei Bedarf durch den Familienmitglied *Ivy* verwalten lassen. Dazu muss *Ivy* in einer separaten Konfiguration eingerichtet werden, die wiederum für den allgemeinen Fall und für jedes Plugin erstellt werden muss. Die Konfiguration enthält die benötigten Bibliotheken, den Versionsrahmen sowie Kriterien zum ein- und ausschließen von transitiven Abhängigkeiten.

Somit erfüllt *Ivy* den Wunsch nach Ordnung und Organisation von Drittanbieter Bibliotheken, jedoch hat diese einen Preis. Die Konfiguration wird in einer separaten XML-Datei verwaltet und gestaltet das Lesen und Nachvollziehen der zusammengeführten Konstruktion von *Ant* und *Ivy* mühselig,

denn das Fehlen einer Konvention, der Zuwachs an den XML-Dateien und ihr globaler Zweck sind für den Entwickler nicht sofort ersichtlich.

Zu diesem Zeitpunkt ist *Ivy* nicht Teil von *Renew*, daher bleibt die Abhängigkeitsverwaltung ein wünschenswertes Feature.

Von Ant zu Gradle

Wie bereits angedeutet, ist die imperative Konfiguration mit *Ant* unstrukturiert und aufwändig. Aus diesem Grund wurde eine eindeutige Konvention gewünscht, die ein fertiges Gerüst an Ausführungsschritten anbietet und für jeden Entwickler sofort verständlich ist. Der Nachfolger *Apache Maven* hat diese Anforderung erfüllt und führt einen global eindeutigen und integrierten Lebenszyklus ein, der mit Plugins und Ausführungsschritten belegt werden kann. Somit verkörpert *Maven* mit dem *convention over configuration* Ansatz ein fertiges deklaratives Rahmenwerk, das trotz der gewünschten Konvention die angebotene Funktion streng an den Plugin Anbieter bindet und somit das Nachrüsten von zusätzlichen Features einschränkt. Aus diesem Grund war das Bedürfnis nach dem nächsten Nachfolger, der die allgemeingültige Konvention von *Maven* übernimmt und diese mühelos erweitern lässt, groß.

Die Herausforderung mit einer Konvention zu arbeiten und diese bei Bedarf erweitern zu können, hat sich das *Gradle* Projekt gewidmet und verspricht eine flexible und erweiterbare Konvention zum Erstellen von ausführbaren Bibliotheken.

Dafür wurde das in *Ant* und *Maven* eingesetzt XML-Format als die einschränkende Schwachstelle eingestuft, denn dieses eignet sich bestens für die Übermittlung von Datenobjekten und dessen Eigenschaften, jedoch für das Schreiben von einfacher Logik werden hunderte Konfigurationszeilen benötigt. Dementsprechend ist das Einführen von zusätzlicher Logik in dem XML-Format nur schwer möglich. Aus diesem Grund verzichtet *Gradle* auf den Einsatz von XML und führt eine Programmiersprache ein, die Logik kompakt definieren lässt und diese mit dem mitgelieferten Ausführungscode problemlos zusammenführen kann.

Der ausgewählte Kandidat trägt den Namen *Groovy*. *Groovy* ist eine dynamische Programmiersprache, die für der virtuellen Maschine von Java entwickelt worden ist und erlaubt an Ort und Stelle das Erweitern und Manipulieren von Objektverhalten.

Dementsprechend stellt *Gradle* eine Lebenszyklus zur Verfügung, der mit Funktionalität belegt werden kann und erlaubt zusätzlich das Manipulieren und Erweitern von Verarbeitungsschritten, ohne die Einführung von zusätzlichen Klassen. In der Konsequenz erfüllt *Gradle* die gewünschte Kombination aus Konvention und Flexibilität.

Abgeschlossenen Build-Umgebung

Das Problem der einfachen Manipulation einer gegebenen Ausführungsstruktur war ein wichtiger Beweggrund für die Entwicklung von *Gradle*, dennoch gab es weitere Wünsche, die das Arbeiten mit Fremdcode erleichtern sollen. Zum Beispiel eine Konvention für die Build-Umgebung einer Applikation, die vom Entwickler konfiguriert, veröffentlicht und eine betriebsfähige *out of the box* Routine anbietet. Dieser Wunsch wurde durch den *Gradle-Wrapper* erfüllt, der eine eigene *Gradle* Version mit sich bringt, die im Weiteren für das Nachladen und Platzieren der benötigten Bibliotheken auf dem Klassenpfad kümmert. Somit ist das Laden und Bauen einer Applikation nicht mehr an die lokale Maschine gebunden und benötigt kein Java oder Drittanbieter Werkzeuge für das Erstellen eines ausführbaren Ergebnisses.

Renew und Gradle

Zusammengefasst vereinigt *Gradle* das Beste aus *Ant* und *Maven*, wie zum Beispiel die Flexibilität von *Ant*, die Konvention sowie die Bibliotheksverwaltung von *Maven* und erweitert diese durch eine Umgebungskonvention sowie eine adaptive Domäne spezifische Sprache, die das Schreiben von Code sowie das Erweitern und Konfigurieren von Abarbeitungsschritten mühelos gestaltet. Des Weiteren werden mit einer abgeschlossenen Umgebung das Bauen und Nutzen von komplexer Software wesentlich einfacher, denn diese ist präpariert und benötigt keine Intervention des Nutzers.

Im Folgenden werden Gründe für RENEW's Umstieg auf *Gradle* zusammengefasst.

- RENEW kann nun auf einen Lebenszyklus mit allen notwendigen Schritten für die Erstellung einer ausführbaren Java Applikation zugreifen und macht somit die manuelle Erstellung von allgemeinen Schrittsequenz wie Kopieren, Kompilieren und Verpacken aus dem *ant* Verzeichnis überflüssig.
- Die importierten Verarbeitungsketten können individuell beeinflusst und erweitert werden, um maßgeschneiderte Ausführungsschritte zu erstellen oder bestehende Schritte auszubauen. Somit können spezifische und notwendige Schritte, wie zum Beispiel das Generieren von Java Klassen, in den bereits vorkonfigurierten Lebenszyklus unkompliziert an der zweckmäßigen Stelle integriert werden. Somit hält die individuelle Plugin Konfiguration nur von der Norm abweichende Logik, die das Lesen und Verstehen der Plugin Funktionalität einfach und unproblematisch gestaltet.
- Mit dem *Gradle-Wrapper* lässt sich eine fertige Build-Umgebung erstellen, die alles Notwendige für das Kompilieren von RENEW in sich

trägt. Somit ist die Installation von *JavaCC*, *Ant*, *Cobertura* und anderer unterstützender Software nicht mehr notwendig.

- Die integrierte Abhängigkeitsverwaltung standardisiert die benötigten Drittanbieter Bibliotheken und minimiert den Verwaltungsaufwand für alle Plugins.
- Des Weiteren gewinnt das *Auschecken* von RENEW Code an Performance, denn die Drittanbieter Bibliotheken sind nicht mehr Teil des Projekts, sondern werden beim Bauen aus dem Web nachgeladen.

5.2.4 Plugin Charakteristika

Jedes RENEW Plugin bietet eine Funktion an, die in bestimmten Einsatzszenarien genutzt werden kann. Um die Funktion anderer Plugins zu nutzen und eigene Funktion anzubieten, wird bestimmte Informationen erwartet, die den Namen, den Zweck und die Basis des Plugins beschreibt. Die benötigte Information wird in der *plugin.cfg* aufbewahrt und von dem RENEW Plugin Manager ausgelesen, um die Plugins in Relation zueinander zu setzen und entsprechend der Aufbaureihenfolge zu instanziieren.

Mit der Einführung des Modulsystems können die Plugins in einer Modulrepräsentation betrieben werden. Dementsprechend wird das Plugin mit einer *module-info.java* erweitert, die sich mit der *plugin.cfg* in der Funktionalität überschneidet. Im Folgenden werden die Schwächen der *plugin.cfg* gegenüber der *module-info.java* untersucht und zusammengefasst.

Die Schwäche der *Plugin.cfg*

Obwohl die *plugin.cfg* die Beziehung der Plugins untereinander beschreibt, ist diese Informantin lückenhaft, denn der *plugin.cfg* aus der Abbildung ?? fehlt die Information über explizite Schnittstellen des Plugins und dafür ausgelegten Pakete. Die *plugin.cfg* besitzt nur eine Auflistung von Plugin Namen, die für die Ausführung präsent sein müssen. Infolgedessen greift der Entwickler auf Funktionalität zu, die nicht für ihn bestimmt ist und beeinträchtigt somit die Wartbarkeit des Systems. Denn jede Änderung der Plugin Logik, kann zu Fehlern und unerwarteten Verhalten seiner Nutzer führen.

Auch wenn die Plugin Abhängigkeit in der *plugin.cfg* existieren, beinhalten

```
name = Renew Formalism
mainClass = de.renew.formalism.FormalismPlugin
provides = de.renew.formalism
requires = de.renew.util,de.renew.simulator
```

Abbildung 5.3: Die *plugin.cfg*

die Konfigurationsdatei keine Drittanbieter-Bibliotheken, die ein wichtiger Bestandteil der Software abbilden. In der Konsequenz kann RENEW nicht

automatisch nachprüfen, ob alle benötigten Drittanbieter-Bibliotheken integriert und mitgeliefert worden sind, um die Software für die Ausführung zu validieren.

Des Weiteren besitzt die *plugin.cfg* ausschließlich der Information über die Laufzeitumgebung, die sich gegenüber der Entwicklungsumgebung unterscheiden kann. Denn, die Entwickler arbeiten auf einer bestens eingerichteten Maschine mit allen nötigen Bibliotheken und Abhängigkeit, die in der Zielumgebung zu meist nicht existieren. Um der Diskrepanz des Ausführungs-kontextes entgegen zu wirken, wird eine Richtlinie benötigt, die den Kontext einer Applikation beschreibt. Leider wird dieses Verhalten nicht von der *plu-gin.cfg* unterstützt.

Die Stärken der *module-info.java*

Die Java Module wurden entwickelt, um die Codebasis beherrschbar, flexibel und austauschbar zu gestalten. Demzufolge verfolgt der Modularisierungsansatz und das Plugin Konzept eine ähnliche Herangehensweise. Der Modularisierungsansatz nutzt die *module-info.java* Konfigurationsdatei, die die Information aus der *plugin.cfg* aufgreift und erweitert.

```
module de.renew.formalism {
    // api offer
    exports de.renew.formalism.function to
        de.renew.misc,
        de.renew.feature.structures;
    exports de.renew.formalism.java;
    exports de.renew.formalism;
    // library demand
    requires junit;
    requires transitive de.renew.simulator;
}
```

Abbildung 5.4: Die *module-info.java*

Der Unterschied zwischen der *module-info.java* und der *plugin.cfg* wird bei der ersten Betrachtung sofort ersichtlich, denn die *module-info.java* schränkt den Zugriffsraum auf das Modul ein, indem nur ausgewählte Schnittstellen-pakete geöffnet werden, die darüber hinaus von jedem oder nur von auserwählten Konsumenten genutzt werden können. In der Abbildung ?? wird die Funktion des *formalism* Plugins über das Paket *de.renew.formalism.function* für die RENEW *misc* und *feature structure* Plugins freigegeben.

Somit enthält jedes Plugin-Modul einen verpflichtenden Kommunikations-vertrag, mit dem das Modul mit seiner Umgebung kommuniziert.

Die verpflichtende Eigenschaft ist neu für *Renew*, da die *plugin.cfg* bislang nur eine optionale Beschriftung trägt und Fehler oder Mängel zu einem späten Zeitpunkt ungünstig auftreten lässt. Um dem entgegen zu wirken, greift Java direkt in das Projekt ein und führt die benötigten statischen Analysen

durch, bevor die Applikation kompiliert und ausgeführt wird. Die Inspektion ist in Java integriert und stellt sicher, dass alle Abhängigkeiten zur Kompilations- sowie Laufzeit verfügbar sind und keine von diesen eine Zyklus hervorruft.

Die Validation mit der *plugin.cfg* ist leider nicht möglich, da sie nicht Teil der Java Plattform ist und nur von dem RENEW Plugin Manager zur Laufzeit verarbeitet wird. Des Weiteren unterstützt die *module-info.java* transitive und statische Abhängigkeiten, die flexible Modul-Zugriffsrechte für die Nutzung weiterreichen und bestimmte Module für optional deklarieren. Dementsprechend wird der Kontext eines Plugin-Moduls implizit an seine Nutzer weitergereicht und macht das Lesen der benötigten Abhängigkeiten verständlich. Denn, die explizite Deklaration jedes benötigten Plugin-Moduls und der dazugehörigen Drittanbieter-Bibliotheken, kann zu hunderten Einträgen führen, die an der Stelle keinen Beitrag zu der Struktur leisten.

Direkter Vergleich

Im Folgenden wird eine Tabelle mit Eigenschaften abgebildet, die Gemeinsamkeiten und Unterschiede der beiden Konfigurationsdatei darstellt.

module-info.java	plugin.cfg
Plugin Abhängigkeiten	Plugin Abhängigkeiten
Bibliothek Abhängigkeiten	
Entwicklung- und Laufzeitanalyse	Laufzeitanalyse
Wesentlich	Redundant
Funktionangebot	Codeangebot
Plattform Verpflichtend	Plattform Optional
Einschränkender Zugriff	Offener Zugriff
Gekapselt und Eindeutig	Gemischt und Verteilt
Integrierte Java Services	Renew Services
Namen	Namen
	Beschreibung
	Eigenschaften
	Version

5.2.5 Service Loader

Der RENEW Plugin-Manager ist der Kern der Applikation und verbindet alle Plugins und erforderliche Drittanbieter-Bibliotheken miteinander. Die geschickte Umsetzung entkoppelt Komponenten voneinander und erlaubt das einfache Hinzufügen sowie Entfernen von Plugins, ohne die bestehende Architektur zu verändern. Die notwendigen Werkzeuge, die im Grundlagenkapitel besprochen wurden, ermöglichen die Umsetzung der dynamische Plugin Kopplung.

Obwohl die Plugins Unabhängig voneinander entwickelt werden können, muss te die interne Funktion an die umgebenen Plugins durch direkte Zugriffe gebunden werden. Daraus folgt eine wilde Kopplung jedes einzelnen Plugins mit der umgebenden Logik. Zum Beispiel greift das *Gui*-Plugin direkt auf den *Formalism*- sowie den *Simulator*-Plugin zu und manipuliert somit den Zustand der Applikation. Im Gegensatz dazu manipuliert das *CNFormalism*-Plugin seinen Zustand direkt über das *CH.if.draw-UI*-Framework. In der Konsequenz ergeben sich widersprüchliche Kontrollflüsse.

Um die Kontrolle über alle möglichen Plugin Typen zu behalten, kann das *Gui*-Plugin über den *Java Service Loaders* Mechanismus alle eingebundenen *Formalism*-Plugins auslesen. Dafür wird ein *Formalism*-Schnittstellen Modul erstellt, welches anschließend von anderen Modulen implementiert werden kann. Zum Beispiel können die *CNFormalism* und das *FAFormalism*-Plugin's, die als Module implementiert sind, mithilfe des *provide with* Schlüssels eine Implementation für das *Formalism*-Modul anbieten. Diese werden anschließend über die *Java Service Loader Registry* von dem *Gui*-Plugin ausgelesen und verwaltet.

Die Instanziierung der Plugins über das *IPlugin* Interface verfolgt einen ähnlichen Ansatz und war somit seiner Zeit voraus. Jedoch hat Java aufgeholt und bietet eine leichtgewichtige und native Umsetzung des Registrierdienstes mit einer variablen Menge an möglichen Interface Implementationen von der RENEW profitieren kann.

5.2.6 Plugin Management

Eine Plugin-Architektur definiert eine Erweiterungsspezifikation, die externen Code in die Applikation lädt und diese zu einem später Zeitpunkt entfernt, ohne alle Details der Codebasis im Voraus zu kennen [?]. Somit müssen die Plugins einem Kommunikationsvertrag eingehen und befolgen, um eine Kommunikation mit der Kernapplikation aufnehmen zu können. Der Kommunikationsvertrag wird mit einem *interface* umgesetzt, auf das sich beiden Kommunikationspartner einigen. Das UML-Diagramm ?? veranschaulicht die Beziehung zwischen den Plugins und dem Plugin-Manager der Kernapplikation.

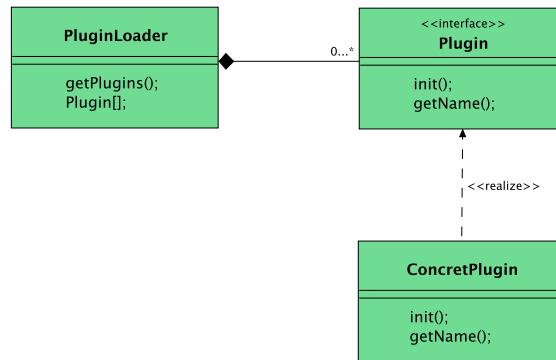


Abbildung 5.5: UML Diagramm für das Plugin-Muster [?]

Der *Plugin Manager* ist zuständig für das Erfassen sowie Einlesen der Plugin Codebasis in dem Arbeitsspeicher und für die Verwaltung des dazugehörigen Lebenszyklus mit allen möglichen Plugin Zuständen während der Laufzeit. Für aufeinander basierende Plugins ist die Struktur strikt, da die Plugins auf bestimmten Drittanbieter sowie Plugin-Bibliotheken aufgebaut sind und diese für die Ausführung benötigten. Wenn die Einlese Reihenfolge der Plugins nicht stimmt oder erwartete Plugins nicht präsent sind, vermisst der Klassenpfad die von dem Plugin angeforderte Klasse und führt zu einem Startabbruch der gesamten Applikation. Daher ist das Laden der Drittanbieter-Bibliotheken sowie der Plugins in der vorgegebenen Reihenfolge eine verantwortungsvolle Aufgabe.

Das Klassenlader System von RENEW

Die Umsetzung der RENEW Plugin Architektur ist durch ein Klassenlader System umgesetzt, welches den RENEW Code in drei Gruppen aufteilt und durch separate Klassenlader in die Applikation aufnimmt. Die Klassenlader verfolgen den empfohlenen und integrierten *parent first* Ansatz, der in dem Kapitel ?? diskutiert wurde und delegiert somit jede Anfrage zuerst an seinen übergeordneten Klassenlader bevor die Klassensuche selbstständig durchgeführt wird.

In der ersten Klassenlader Instanz wird die Basis etabliert, die den Plugin-Manager sowie die Drittanbieter-Bibliotheken einliest und den darunter liegenden Plugin-Klassenlader mit Funktion versorgt. Der Plugin-Klassenlader lädt anschließend alle Plugins nach einander in der richtigen Reihenfolge ein und garantiert zur Laufzeit das Vorkommen jeder benötigten Plugin-Klasse auf dem Klassenpfad. Zuletzt wird ein dynamischer Klassenlader erstellt, der für das Simulieren der Petrinetze entsprechende Klassen und Ressourcen beim Ausführen einliest und nach der Simulation vergisst.

Die Abbildung der ausformulierten Klassenlader Systems von Michael Duvi-

gneau wird in der Abbildung ?? dargestellt.

Duvigneaus Java Prototyp [?] sollte eine Plugin-Verwaltung anbieten, die

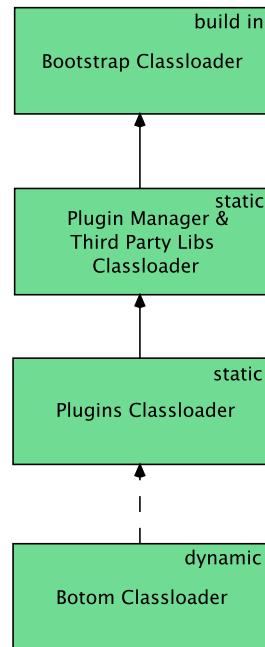


Abbildung 5.6: RENEW Klassenlader System Umsetzung von Duvigneau [?]

Plugins einfach sowie dynamisch neu konfigurieren lässt und das Hinzufügen und Entfernen von Plugins während der Laufzeit ermöglicht. Da der Prototyp nur die Praxistauglichkeit demonstrieren sollte, wurde eine schnellere Variante implementiert, die nur die nötigste Funktionalität bereitstellt. Die Umsetzung der Klassenlader-Architektur aus der Abbildung ?? hat sich als ausreichend, jedoch nicht Mängelfrei herausgestellt. Die Umsetzung unterstützte ein essenzielles Charakteristikum eines dynamischen Systems nicht, nämlich das Entladen der Plugin Codebasis aus der Applikation während der Laufzeit. Denn, die Plugins werden mit einem gemeinsamen Klassenlader in das System eingebunden, der keine Möglichkeit anbietet eine auserwählte Codebasis zu vergessen. Dementsprechend können Plugins, die mit einem gemeinsamen Klassenlader geladen worden sind, nicht mehr separat behandelt werden. Dies hat zur Folge, dass die Applikation während der Laufzeit ständig wächst und die Möglichkeit verliert Plugins während der Laufzeit zu aktualisieren. Somit ist man an einen Neustart angewiesen, der die initiale Codebasis zurücksetzt und die Plugins entfernt oder aktualisiert. Daraus folgt eine erhebliche Beeinträchtigung der Wartbarkeit und der Systemstabilität für Systeme mit einem permanenten Betrieb.

Einer der vorgeschlagenen Lösungsansätze ist das Erstellen eines Klassen-

lader per Plugin und die Aufbereitung sowie Verwaltung der Kommunikation zwischen diesen. Da der mitgelieferte Klassenlader von Java nur einen übergeordnet Klassenlader akzeptiert, erfordert die Umsetzung eine eigene Klassenlader Implementation, die variable Eltern Kontenmenge akzeptiert und die Delegation intelligent umsetzt.

Die genannte Umsetzung bedarf einen enormen Aufwand und viel Eigenimplementation. Eine Alternative bietet das *OSGi* Rahmenwerk, welches auf dynamische Systeme ausgelegt ist und eine Komponentenbasierte Entwicklung ermöglicht. Das *OSGi* Rahmenwerk organisiert den Code in *bundles*, die sich auf separaten *OSGi* Klassenlader Implementation befinden und über eine Verknüpfung der *bundles* eine Applikation modellieren.

Die Umsetzung von *OSGi* im RENEW Kontext hat sich für schwierig erwiesen, da die Verwaltung der Plugins als *OSGi bundles*, mit einem erweiterten Lebenszyklus und Kommunikationsschwierigkeiten, den gewünschten Anforderungen nicht entsprach [?]. Demzufolge wurde die Java Umsetzung des Klassenlader Systems der *OSGi* Umsetzung vorgezogen und bis zum jetzigen Standpunkt einwandfrei betrieben.

Mit der Einführung des Modulsystem von Java, ist Java fähig das genannt Problem anzugehen und bietet eine Alternative zu den beiden genannten Lösungsansätzen. Die Umsetzungsmöglichkeit der dynamischen Plugin Verwaltung wird im nächsten Abschnitt der neu eingeführten Modulschichten diskutiert.

Plugin Management mit Modulschichten

Die grundlegende Idee der Modulschichten wurde im Kapitel ?? diskutiert. Mithilfe der Modulschichten wird in diesem Abschnitt ein Grundriss des möglichen Einsatzes im Plugin-Management vorgestellt.

Das Zentralproblem der zurzeit betriebenen Umsetzung, ist das monolithische Verhalten der Plugins, die sich den gleichen Klassenraum teilen, direkte Zugriffe durchführen und während der Laufzeit nicht voneinander getrennt werden können. Denn, die RENEW-Plugins sind an den Plugin-Klassenlader gebunden und dürfen laut der Java Plattform diesen nicht mehr verlassen. Um die Plugins aus der Applikation endgültig zu entfernen, müssen die Plugins auf separate Klassenlader aufgeteilt und betrieben werden. Durch den Einsatz von mehreren Klassenlader kann die Referenz auf einen bestimmten Klassenlader Objekt, welches ein Plugin in sich trägt, gelöscht oder neue belegt werden. Somit ist das ehemalige Klassenlader Objekt nicht mehr erreichbar und wird von dem *Garbage Collector* aus dem Speicher entfernt.

Wie bereits in den Grundlagen Kapitel ?? behandelt, sind die zur Verfügung gestellten Klassenlader Hierarchisch aufgebaut, arbeiten nach dem *Parent*

First-Delegationsprinzip und können die Anfrage nur an einen übergeordneten Klassenlader delegieren. Dementsprechend fehlt dem Klassenlader-System die Integration der verzweigten Delegation sowie die organisierte Kommunikation zwischen einer variablen Anzahl an Klassenlader, die für das Plugins System eine große Rolle spielen.

Mit der Integration der Module sowie Modulschichten in die Java Plattform, sollen die angesprochenen Probleme adressiert und gelöst werden. Aus diesem Grund kümmert sich das Modulsystem von Java um die Authentizität sowie Integrität des Modulinhalts und führt zusätzlich das Konzept der Modulschichten eine, welches das Verwalten der System Modulbausteine übernimmt und den genauen Aufenthaltsort jedes Moduls sowie seiner zuständigen Verwaltungsinstanz überblickt.

Die Verwaltung des Modulinhalts und dessen Abhängigkeiten werden über die expliziten Schnittstellen angegeben und über eine Konfiguration, die einen Teil der Modulschicht darstellt, validiert. Die Konfiguration erstellt einen Modul Abhängigkeitsgraphen, der aus einer gegebenen Verzeichnis Menge, einem auserwählten Wurzelmodul und einer optionalen Anzahl von untergeordneten Konfigurationen Modulbeziehungen herstellt. Die Konfiguration validiert und stellt sicher, dass alle benötigten Abhängigkeiten gefunden und aufgelöst werden, keine Zyklen durch das hinzufügen neuer Module entstehen und jedes Modul eindeutig jeder Suchanfrage zugeordnet werden kann. Somit überblickt die Konfiguration das Modulsetup und den Zustand ihre Schicht sowie der darunterliegenden Modulschichten. Dies hat zur Folge, dass Module nur einmal in der Schicht Konfiguration auftreten können und verhindert das Überladen von Modulfunktionalität.

Die elastische Konfigurationseinstellung und die globale Konsistenz der modularen Applikation ermöglichen eine verzweigte Delegation der Suchanfrage an bekannte Modulschichten und dessen Module, die zuvor Validiert und in die Applikation eingebettet worden sind. Somit kennt die Applikation alle Module sowie die dazugehörige Modulschichten und kann die Suchanfrage an die entsprechende Schicht weiterleiten, ohne ein Sicherheitsrisiko einzugehen.

Die neu eingeführte verzweigte Delegation spielt eine große Rolle für das Plugin Konzept, denn die Plugins erweitern und werden von anderen zahlreichen Plugins erweitert. Das dynamische Verhalten der Plugins kann ohne Struktur und umfassende Sicherheitsmechanismen nicht mängelfrei in einem großen System bestehen. Dementsprechend muss das Erweiterungsmodul von unerwarteten äußeren Einflüssen geschützt werden und vor dem Einbinden in die Applikation eine valide Strukturintegration nachweisen. Die genannten Anforderungen werden bestens von den Modulen und den Modulschichten abgedeckt, die mit der zusätzlichen verzweigten Delegation neue Möglichkeiten der Java Plattform bieten.

Im Folgenden wird ein Prototyp vorgestellt, der ein Plugin System simuliert,

welches aus Modulen sowie Modulschichten besteht und als ein Grundriss für die mögliche Umsetzung der nächsten RENEW Version dienen könnte.

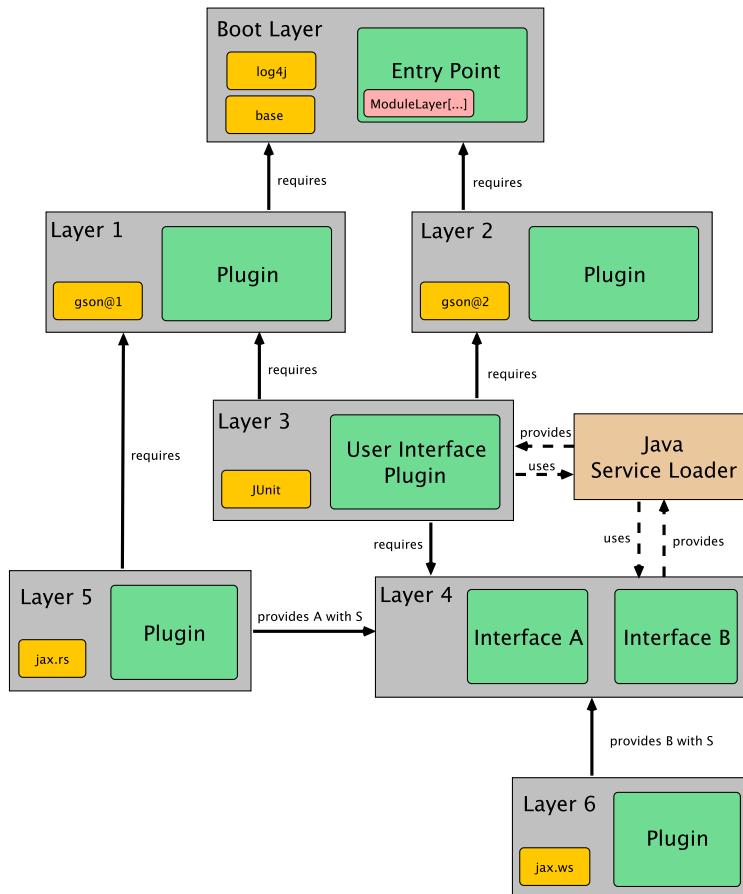


Abbildung 5.7: Grundriss einer möglichen Plugin Architektur auf Basis der Modulschichten

Ganz Oben in der Schichten Hierarchie befindet sich der *Boot Layer*, dieser enthält die von Java zur Verfügung gestellten Module sowie unseren Einstiegspunkt der Applikation. Der Einstiegspunkt der Applikation erstellt und verwaltet die Plugins über die zur Laufzeit erstellten Modulschichten. Die darunterliegenden Schichten *eins* und *zwei* stellen zwei unterschiedliche Plugin Funktionalitäten dar, die dieselbe Bibliothek *gson* in unterschiedlicher Version nutzen und die Möglichkeit des parallelen Betriebs illustrieren. Die Modulschicht *drei* baut auf der *zweiten* und *dritten* Modulschicht auf und benötigt dessen Funktionalität für die eignen Implementation. Zum Beispiel könnte es die Funktion des *GUI-Plugins* darstellen, die die *Util* sowie *WindowManagement* Plugins aus der zweiten sowie dritten Modulschicht anfordert. Dementsprechend kann ein Plugin mithilfe der Modulschichten zwei

oder mehr übergeordnete Schichten referenziert und auf ihre Funktionalität Konfliktfrei zugreifen.

Da das *Gui*-Plugin nicht auf allen RENEW-Plugins aufbaut, sondern diese lediglich Koordiniert, muss die Logik, die in parallelen Modulschichten entstehen, für das *Gui*-Plugin zugänglich gemacht werden. Dies wird mithilfe des Java *ServiceLoader* umgesetzt, der für ein gegebenes Schnittstellen Modul alle zugänglichen Implementationen erfasst und als ein Service für die Ausführung dem *Gui*-Plugin anbietet.

5.3 Anforderungsanalyse

Indem der Java-JDK mit der neunten Version von Java modularisiert wurde, ändern sich die grundsätzlichen Funktionsrichtlinien der Plattform, die Konsequenzen für bestehende Systeme mit sich bringen. Denn das Modulsystem von Java führt eine obligatorische Kapselung der Code-Komponenten ein, die über erweiterte Sicherheitsmechanismen verfügen und nur über explizite Schnittstellen geladen und angesprochen werden können.

Diese Abschlussarbeit beschäftigt sich mit der Untersuchung von Anforderungen der modularisierten Java Plattform, sowie dem entsprechenden Aufwand für das Anpassen bestehender Systeme.

Für die Umsetzung muss ein grundlegendes Migrationsverfahren erarbeitet und angewandt werden, welches eine Migration auf das Modulsystem von Java ermöglicht. Da die Migration bestehender Systeme nicht in einem Schritt durchgeführt werden kann, müssen Charakteristika für Module formuliert werden, um bestehende Systemelemente innerhalb einer ausgereiften Software mit den entsprechenden Eigenschaften zu erweitern.

Jede kleine Änderung in einem komplexen System bringt große Risiken mit sich, die das saubere Arbeiten der Software in Frage gestellt. Daher ist der Übergang auf das Modulsystem von Java mit Unsicherheit verbunden, zumal es Zeit kostet, im Code verankertes Geschäftswissen umstrukturiert und keine sichtbare Softwareerweiterung für den Endkunden darstellt. Demnach liegen die Schwerpunkte der Migration in der Konsistenz der gegebenen Software im neuen Kontext, dem Mehrwert sowie in dem Aufwand der Umsetzung.

Um die gegebenen Anforderungen an eine Applikation zu stellen, bedarf es einer passenden Projektstruktur und Umsetzungswerzeuge, die eine Modulmenge kompilieren und verpacken. Zu diesem Zweck soll das Java basierte Gradle Werkzeug eingeführt werden, das eine kompakte und mächtige Ausdrucksform für das Erstellern von Softwaresystemen Entwicklungsumgebung unabhängig anbietet.

Für die Umsetzung der erarbeiteten Konzepte steht der RENEW Simulator

und das MULAN Rahmenwerk zur Verfügung, die in dieser Abschlussarbeit Migrationsszenarien erleben.

Das erste Szenario soll eine kontinuierliche Migration einer Software auf das Modulsystem modellieren. Im Gegensatz dazu, wird im zweiten Szenario alt Software auf eine modularisierte Code-Basis aufgesetzt und der parallele Betrieb begutachtet.

5.4 Anforderungsspezifikation

Für den Anwendungskontext der Modularisierung einer größeren Anwendung steht RENEW und MULAN zur Verfügung, jedoch übersteigen eine vollwertige Modularisierung von RENEW und MULAN den Zeitrahmen dieser Abschlussarbeit, da die Umstellung jedes einzelne Plugins auf organisierte Modulverbände eine zeitintensive Aufgabe verkörpert. Jedes einzelne Plugin muss analysiert, reorganisiert, auf Module zerlegt und bestens mit einander verzahnt werden. Darüber hinaus müssen die entstandenen Module mit allen gegenwärtigen Plugins abgestimmt werden, da das Plugin Modulverband generell andere Schnittstellen anbieten wird.

Dementsprechend wird die Modularisierung auf einzelne Plugins reduziert, die für die Darstellung der UI sowie die grundsätzliche darunterliegende Logik zuständig sind. Um die Kommunikation im bestehenden System zu garantieren, behalten die entstandenen modularisierten Plugins ihre Schnittstellen. Infolgedessen wird die Funktionalität aller Plugins nicht beeinträchtigt und garantiert dem Gesamtsystem einen nahtlosen Übergang auf den Modulpfad. Hierfür werden die Plugin Module mit den entsprechenden Konfigurationsdateien erweitert und mit den notwendigen Plugins verzahnt. In der Konsequenz wird die Umsetzung nur die minimalen Anforderungen des Modulsystems realisieren und das Zerlegen der internen Struktur der Plugins aus der Sicht lassen, obwohl es ein wünschenswerter Schritt in Richtung der Modularisierung wäre.

Trotz allem müssen die Plugin Projekte von RENEW zusätzliche Module unterstützen und die Möglichkeit bieten, RENEW zu erweitern, um weitere Module innerhalb eines Plugins erstellen zu können. Dafür benötigen alle Plugins eine neue Projektstruktur, die das gewünschte Verhalten umsetzt und die empfohlene Modulorganisation unterstützt. Dieses trägt die Maven konforme Projektstruktur, die Java Klassen, Ressourcen und sonstige Artefakte nach einem standardisierten Muster verwaltet.

Des Weiteren muss die Konfiguration des Projektes von der Entwicklungsumgebung gelöst werden, um den Entwickler nicht in seinem Arbeitsablauf einzuschränken und die Entwicklungsumgebung seiner Wahl zu nutzen. Dazu soll die existierende Ant Umgebung mit Gradle ersetzt werden. Diese erleichtert das Verwalten sowie Verpacken der Software und verspricht zusätz-

lich eine kohärente Arbeitsweise mit dem Modulsystem von Java, indem das Verwalten der Modulabhängigkeiten sich in beiden Systemen widerspiegeln. Diese Eigenschaft soll auf die modularisierte Version von RENEW angewandt und evaluiert werden.

Da der RENEW Simulator von verschiedenen Abschlussarbeiten beleuchtet und erweitert wird, beinhaltete die Applikation verschiedene Techniken der Softwareentwicklung, wie zum Beispiel das Generieren von Java Klassen mit dem *JavaCC* Compiler. Dementsprechend existieren ältere Verfahren innerhalb von RENEW, die sorgfältig auf Kompatibilität mit der neuen Umgebung untersucht und migriert oder ersetzt werden müssen.

Das Aufbereiten der Grundlagen sowie der Ausgangssituation sind essenzielle Schritte einer Migration, da diese das Fundament für die Planung und Umsetzung legen. Zusätzlich leitet die Ausgangssituation das Migrationsszenario, das eine beispielhafte Schritt für Schritt Migration oder Integration demonstriert und evaluiert.

5.5 Ausgangssituation

In diesem Abschnitt wird die Ausgangssituation für RENEW sowie MULAN dargelegt, die als Grundlage für die nachfolgenden Prototypen dienen werden.

5.5.1 Renew

RENEW ist in mehr als 60 Plugins aufgeteilt, die für sich alleinstehende Projekte repräsentieren. Jedes Projekt besitzt eine *build.xml* und wird mit dem übergeordneten Stamm *build.xml* Script zusammengeführt. Die XML-Scripte werden von dem *Apach Ant* Werkzeug evaluiert, kompiliert und zusammengeführt. In Folge dessen entsteht ein *jar* Archiv für jedes Plugin-Projekt. Diese werden in eine bestimmte Orderstruktur für die Ausführung aufbereitet, die sich aus dem *config*, *plugins* und *libs* Verzeichnis zusammensetzt.

Der innere Aufbau jedes Plugins benötigt eine besondere Konfigurationsdatei, nämlich die *plugin.cfg*. Diese beschreibt für die Ausführung nötigen Plugin-Abhängigkeiten und wird von dem internen Plugin-Manager verwaltet, der für die richtige Ordnung beim Laden jedes einzelnen Plugins aus dem *plugins* Verzeichnis sorgt. Somit sind die *plugin.cfg* Dokumente ein guter Startpunkt für die Evaluation einer minimalen und lauffähigen RENEW Konfiguration.

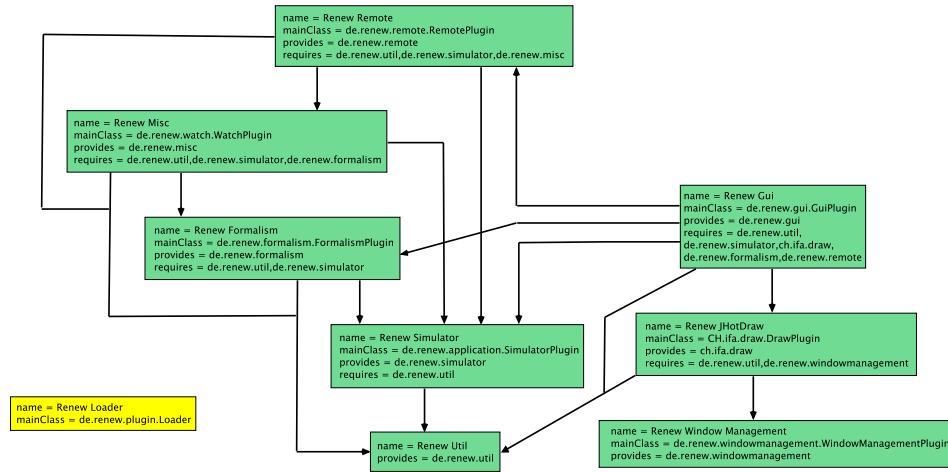


Abbildung 5.8: Gui Plugin-Abhängigkeiten

Für die Evaluation der minimalen Konfiguration starten wir aus dem *Gui*-Plugin und arbeiten uns abwärts der Plugin-Hierarchie der *plugin.cfg*'s hinab, bis der komplette Graph aufgebaut ist.

Die in der Abbildung ?? repräsentierten Zusammenhänge reflektieren die von den Entwicklern abgestimmten Laufzeitabhängigkeiten, die einen groben Überblick über die nötigen Plugins verschaffen. Diese können zur Laufzeit alle benötigten Daten und Klassen enthalten, jedoch tragen sie keine Aussage über Abhängigkeiten während der Kompilation. Demgemäß kann zusätzlicher Code sowie Plugins benötigt werden.

5.5.2 Mulan

MULAN [?] ist ein Multiagenten Rahmenwerk, mit dem Agenten entwickelt und miteinander verbunden werden können. Diese agieren nach eigenem Interesse und verhandeln über Kommunikationskanäle, die von MULAN angeboten werden. Um eine auf MULAN basierende Multi-Agent-Anwendung zu erstellen, müssen zahlreiche Protokollnetze gezeichnet und Wissensbasen erstellt werden, die das Verhalten und die Interaktion von Agenten implementieren. Somit bietet MULAN eine Kommunikationsplattform sowie ein Gerüst für die Umsetzung der darauf aufsetzenden Agenten und dessen Eigenschaften sowie Fähigkeiten, die mit Referenznetzen entworfen werden können. [?]

Das Spiel Settler ist mit MULAN umgesetzt und implementiert somit die Agenten Anforderungen des Multi Agenten Systems. Diese halten Wissensbasen über die Spielregeln und können mit Hilfe der Protokollnetze bestimmte Entscheidungen treffen und Handlungen ausführen. Wie zum Beispiel Straßen bauen oder mit Karten handeln.

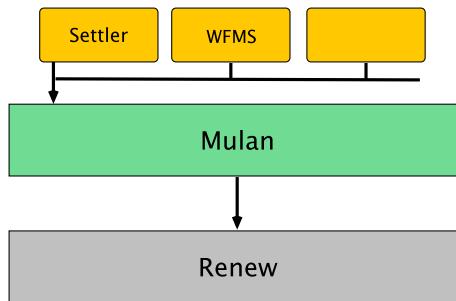


Abbildung 5.9: MULAN Plugins

Damit die erstellten Settler Agenten und dessen Aufbau simuliert werden können, wird der RENEW Petrinetz Simulator verwendet, mit dem die umgesetzten Agenten-Strukturen betrieben werden können.

Des Weiteren besitzt das Settler Spiel sowie das MULAN-Rahmenwerk eine Plugin-konforme Architektur und werden somit genauso, wie die RENEW Plugins von dem RENEW Plugin-Manager ausgelesen und verwaltet.

Ein vereinfachter Zusammenhang ist in der Abbildung ?? dargestellt und visualisiert die benötigten Grundlagen für die Ausführung von Settler.

5.6 Durchführung

Die wichtigen Szenarien für das Erreichen des gesetzten Ziels werden mit zwei Prototypen umgesetzt. Der RENEW Prototyp wird eine kontinuierliche Modularisierung der Plugins und dessen Betriebsfähigkeit mit nur zum Teil Modularisierter Codebasis abdecken. Dazu gehört das Erstellen einer neuen Projektstruktur für die auserwählten Plugins und behandelt auserwählte Richtlinien, die mit dem Modulsystem von Java eingeführt worden sind. Für das Anfertigen von ausführbar Code, wird das Gradle Werkzeug integriert, das einen Kompilation Kontext für die neuen Module erstellt. Dazu gehören eine Abhängigkeitsverwaltung sowie Projektkopplung. Um die Plugins mit einander als Module zu verzähnen, werden Schnittstellen untersucht und in der *module-info* Konfigurationsdatei in den Plugins verankert. In der folgenden Evaluation wird das Ergebnis evaluiert und sauber Aufbereitet.

Im Gegensatz dazu behandelt der MULAN Prototyp die Zusammenführung bestehender Systeme mit modularisiertem Code. Für die Umsetzung werden nur für das Spiel Settler relevanten MULAN Plugins erarbeitet und auf die minimale RENEW Version aufgesetzt. Doch zuerst muss eine erweiterte Analyse der minimalen RENEW Version durchgeführt werden, um die benötigten RENEW Plugins für das MULAN Rahmenwerk nachzurüsten. Im weiteren Verlauf behandelt dieser Prototyp Konsequenzen der Umstellung

5.6. Durchführung

auf eine neue RENEW Grundlage und der benötigten Ausführungsschritte für die Adaption. Die Evaluation bewertet den Aufwand für die Umsetzung des Prototyps.

Kapitel 6

RENEW Prototyp

In diesem Kapitel entsteht ein Prototyp, der RENEW schrittweise modularisiert, bis die Applikation den größten Teil ihre Funktionalität auf dem Modulpfad betreiben kann.

Für die Umsetzung des Prototyps werden zuerst Anforderungen erfasst, die der modularisierte RENEW Prototyp erfüllen muss, um unserer Vision der Implementation zu entsprechen. Infolgedessen entsteht ein Implementierungsplan sowie ein Prototyp.

6.1 Anforderungen

Im Kern der Modernisierung von RENEW liegt die Anpassung von RENEW an das Modulsystem von Java und dessen Anforderungen an Applikationskomponenten. Aus den RENEW Plugins sollen explizite Module entstehen, die auf dem Modulpfad betriebsfähig sein müssen. Die Drittanbieter-Bibliotheken sollen mit in den Modulpfad aufgenommen werden und als automatische Module ihre Aufgabe erfüllen. Zusätzlich darf die Migration und damit verbundene Anpassung und Aufbereitung der Mängel die Kommunikation sowie interne Funktionsweise von RENEW nicht verändern. Dementsprechend soll garantiert werden, dass die darunter liegende theoretische Grundlage in Takt bleibt.

6.1.1 Interaktion

Der erste modulare RENEW Prototyp soll mit einer minimalen Plugin Anzahl auf dem Modulpfad betriebsfähig sein und eine Möglichkeit bieten, Petrinetze zu erstellen, zu simulieren und zu serialisieren. Das heißt, es muss eine UI zu sehen sein, die mit den nötigen Werkzeugen und der darunter liegenden Logik ausgestattet ist.

6.1.2 Projektstruktur

Für die Umsetzung des modularen RENEWS wird für jedes Plugin eine moderne Projektstruktur benötigt, die den Inhalt entsprechend dem etablierten Maven Standardverzeichnislayout auf Java Module und die dafür benötigten Ressourcen aufteilt.

6.1.3 Entwicklungsumgebung

In der existierenden RENEW Entwicklungsumgebung werden alle Plugin Projekte durch eine versteckte `.project` beschrieben. Das heißt, der Klassenpfad und die Bindung der Codebausteine geschehen versteckt und für den Entwickler schwer zugänglich. Damit ist der Entwickler gezwungen, den weiten und verschachtelten Weg durch die UI Konfiguration von Eclipse zu betreten, der sich mit der Zeit wandeln kann. Dieser Sachverhalt wurde von mir im letzten Projekt beobachtet und kostete Zeit für alle Projektteilnehmer, da die Universitätsrechner strikten Rechten unterliegen, die keine Benutzer definierte Eclipse Entwicklungsumgebung aufsetzen lässt. Darüber hinaus ist die Konfiguration von RENEW in anderen Entwicklungsumgebungen wie IDEA oder Netbeans mit der `.project` Konfigurationsdatei nicht möglich.

Um eine Entwicklungsumgebung unabhängige Konfiguration anzulegen, wird ein neues Werkzeug benötigt.

6.1.4 Packaging

Da RENEW an das Modulsystem angepasst werden muss, muss die Prozedur für das Kompilieren und das Verpacken der Codebasis die Veränderung mit erleben.

RENEW benutzt zurzeit das *Apache Ant* Werkzeug, das alle Plugins kompiliert und in eine ausführbare Form bringt. Dieses ist in Jahre gekommen und enthält wesentlich geringeren Funktionsumfang gegenüber der aktuellen Konkurrenz, wie Maven und Gradle. Sie bieten eine Abhängigkeitsverwaltung, konfigurierbare Plugins und Programmiersprachen. Im Gegensatz zu der aufgeblasenen XML-Konfiguration von Ant, die jeden kleinen Schritt ausführlich dokumentiert, beherrschen die modernen *build* Werkzeuge die Komplexität durch den *Convention over Configuration* Ansatz und flexiblen Ausdrucksweisen.

Die minimale Version von RENEW soll sich an einem modernen *build* Werkzeug bedienen und ein ausführbares Ergebnis erzielen.

6.2 Spezifikation

Um die Anforderungen umzusetzen, wird die erarbeitete minimale Version isoliert, umstrukturiert und mit dem Gradle *build* Werkzeug für das Arbeiten in der Entwicklungsumgebung IDEA aufgerüstet. Da Gradle die Verwaltung des Projekts sowie das Kompilieren und Erstellen von ausführbaren Paketen übernehmen kann, ist es eine gute Wahl für das Aufsetzen einer modernen modularen Projektstruktur.

Dafür muss das bestehende Ant *build* System analysiert und mit dem Gradle Werkzeug wiederaufgebaut werden. Dieses soll so gut wie möglich die bestehende Drittanbieter-Bibliotheken verwalten, Module kompilieren und die benötigten Erweiterungen, wie das JavaCC Werkzeug, unterstützen.

Nachdem die Projektstrukturen die passende Form angenommen haben, müssen die Projekt Abhängigkeiten analysiert und innerhalb der *module-info.java* aufgenommen werden.

Zu Letzt entsteht eine bekannte Ordnerstruktur mit Drittanbieter-Bibliotheken, Plugins und Konfigurationsdateien, die mithilfe des *Plugin Managers* verwaltet werden.

6.3 Entwurf

Der Entwurf berücksichtigt die schrittweise Migration und lässt die RENEW Applikation während der Gesamtmigration betriebsfähig bleiben. Das heißt, Plugins auf den Klassenpfad sowie Modulpfad können nahtlos mit einander kommunizieren und ihre Funktion während der Migration weiterhin erfüllen.

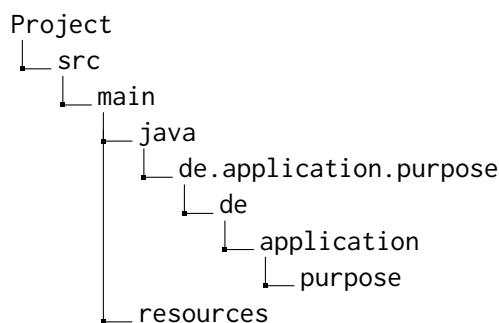


Abbildung 6.1: Projektstruktur

Für den ersten Prototypen wird zuerst eine Projektstruktur erstellt, die für jedes Plugin Projekt die Möglichkeit bieten soll, aus mehreren Modulen zu bestehen. Dafür wird eine Struktur ?? erstellt, die im Java Verzeichnis alle Module bündelt, die über den Modulnamen disjunkt voneinander verwaltet

werden. Nichtsdestotrotz gehören sie zum gleichen Projekt und teilen unter sich das Ressourcen Verzeichnis, das im weiteren Verlauf zum Erstellen der ausführbaren Pakete benötigt wird.

Nachdem die Projektstruktur der gewünschten Form entspricht, muss diese in den Gradle Konfigurationsdateien verankert werden. Hierfür wird für jedes Projekt die Projektstruktur und dessen Abhängigkeiten in der *build.gradle* Konfigurationsdateien ?? festgehalten, indem Java sowie Ressourcen Stammverzeichnisse und Projekte definiert und Drittanbieter-Bibliotheken Abhängigkeiten für den Kompilation-Pfad bestimmt werden.

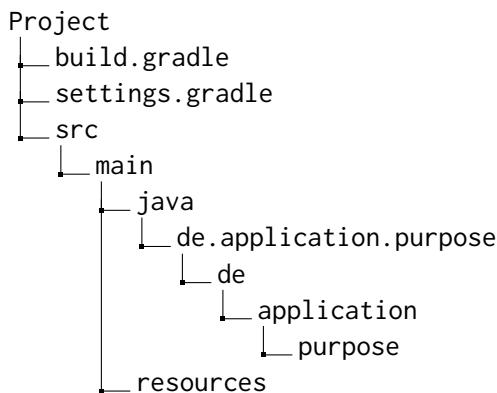


Abbildung 6.2: Gradle Konfiguration

Die oben genannten Schritte müssen für jedes Projekt der minimalen Version von RENEW durchgeführt und im Anschluss über die entsprechende Entwicklungsumgebung validiert werden. Wenn diese alle Klassen und die benötigten Abhängigkeiten finden und kompilieren kann, wurden alle Projekte richtig strukturiert, definiert und miteinander sauber verbunden. In diesem Zustand ist die komplette Struktur des Projekts innerhalb von Gradle verpackt und kann von jeder Entwicklungsumgebung ausgelesen werden.

Da jetzt eine lauffähige minimale RENEW Version für den Klassenpfad erstellt werden kann, ist es Zeit diese zu Modularisieren und die einzelnen Plugins auf den Modulpfad zu migrieren. Dafür werde ich den *bottom up* Ansatz aus dem Kapitel Migration ?? verwenden und Schritt für Schritt die Plugins auf den Modulpfad bewegen.

Zuerst werden die Drittanbieter-Bibliotheken, wie *log4j*, auf den Modulpfad als automatische Module eingebunden und werden damit aus den Klassen- sowie Modulpfaden für die Nutzung zugleich erreichbar sein. Anschließend werden Plugins als explizite Module migriert, die keine Plugin Abhängigkeiten besitzen und aus dem Modulpfad keine Zugriffe auf den Klassenpfad ausführen müssen. Beispielsweise besitzt das *Util* Plugin keine Abhängigkeiten

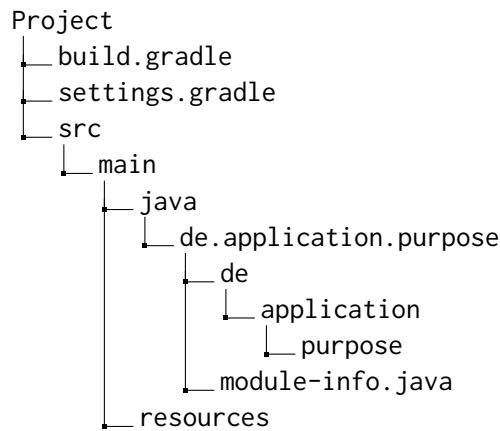


Abbildung 6.3: Modulumwandlung

auf RENEW Plugins und wird für die Ausführung auf dem Modulpfad, durch eine *moduel-info.java* Konfigurationsdatei erweitert. Wie in der Abbildung ?? dargestellt, muss sich diese im Stammverzeichnis des Moduls befinden und die benötigten automatischen Module deklarieren.

In den nächsten Schritten werden Plugins Schritt für Schritt auf den Modulpfad migriert, indem für jedes Plugin eine eigene *module-info.java* Konfigurationsdateien angelegt wird, in der sich ihre Abhängigkeiten auf automatische Drittanbieter-Module sowie explizite Plugin-Module befinden.

Dieses Vorgehen wird solange durchgeführt bis jedes Plugin sich auf dem Modulpfad befindet.

6.4 Umsetzung

Die Umsetzung realisiert den Entwurf und erstellt eine neue Projektstruktur für alle Plugins der minimalen RENEW Version ?. Diese werden anschließend mit dem Gradle Werkzeug zusammengeführt und bilden ein kompilierfähiges Konstrukt. Nachfolgend werden die ausgewählten Plugins, mit der *module-info.java* versehen und deklarieren innerhalb der Konfigurationsdateien die zuvor vorgestellten Kommunikationskanäle ?? zwischen den Plugins.

Der Modularisierungsprozess geschieht iterativ und wird für jedes Plugin einzeln nacheinander durchgeführt.

6.4.1 Umstrukturierung

Für die Umsetzung der Anforderungen und des Entwurfs wird zuerst die Projektstruktur jedes Plugins angepasst. Dafür wird die Struktur jedes Plugins analysiert, umstrukturiert und im weiteren Verlauf von Mängeln befreit. In den meisten Fällen werden *split packages* und gemischte Strukturen innerhalb der Codebasis erwartet.

Zuerst wird eine grobe Maven Projektstruktur erstellt, in dem sich zusätzlich ein Plugin Wurzelverzeichnis befindet. Anschließend migriert man die Codebasis in das Plugin Wurzelverzeichnis, welches den neuen Plugin Namen trägt.

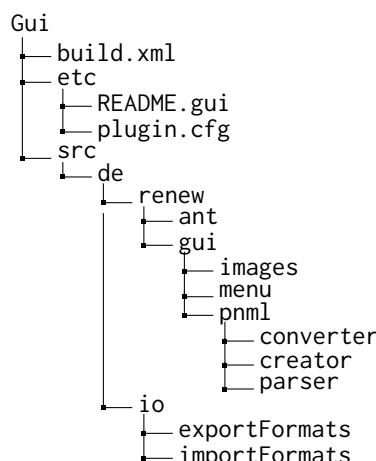


Abbildung 6.4: Gui Projekt

Das Gui Plugin enthält die geläufige mangelnde Organisation der Ressourcen. Zum Teil befinden sich diese in den *etc* Verzeichnis und zum Teil sind diese in dem Java *source set* Verzeichnis integriert, wie in der Abbildung ?? dargestellt.

Um diese zu beheben, wird das Verzeichnis *de.renew.gui.images*, das mit den *png* und *gif* Daten gefüllt ist, in das Ressourcen Verzeichnis migriert. Damit die Applikation diese wiederfindet, werden die Zugriffspfade für die Ressourcen innerhalb des Gui Plugin an das neue Verzeichnis angepasst, indem die internen statischen Konstanten, wie *CPNIMAGES*, auf den entsprechenden Ort verweisen.

Zum Schluss werden die *README* und die *plugin.cfg* aus dem *etc* Verzeichnis in das Ressourcen Verzeichnis bewegt. Somit ist eine Struktur erstellt worden, die sich auf das Modulsystem von Java anwenden lässt.

Andere Plugins wie Formalism, CH oder Misc besitzen *JavaCC* Dateien, wie im Beispiel ?? dargestellt, enden diese auf *.jj*. Sie erstellen Java Netz Grammatiken und wandeln die Java-Basis für die Ausführung ab. Diese liegen lose

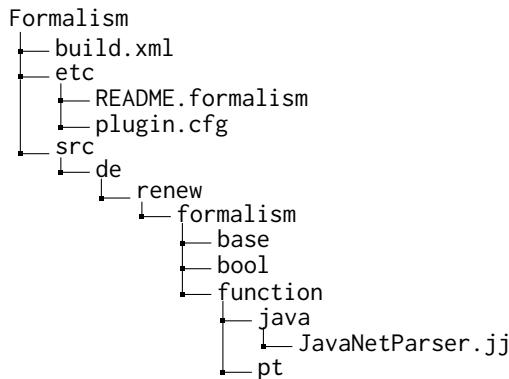


Abbildung 6.5: Formalism Projekt

zwischen den Java Klassen und werden von den Java Compiler nicht interpretiert. Daher macht es Sinn, diese in ein eigenes *Source Set* auszulagern und für den JavaCC Compiler für die Übersetzung zu gruppieren.

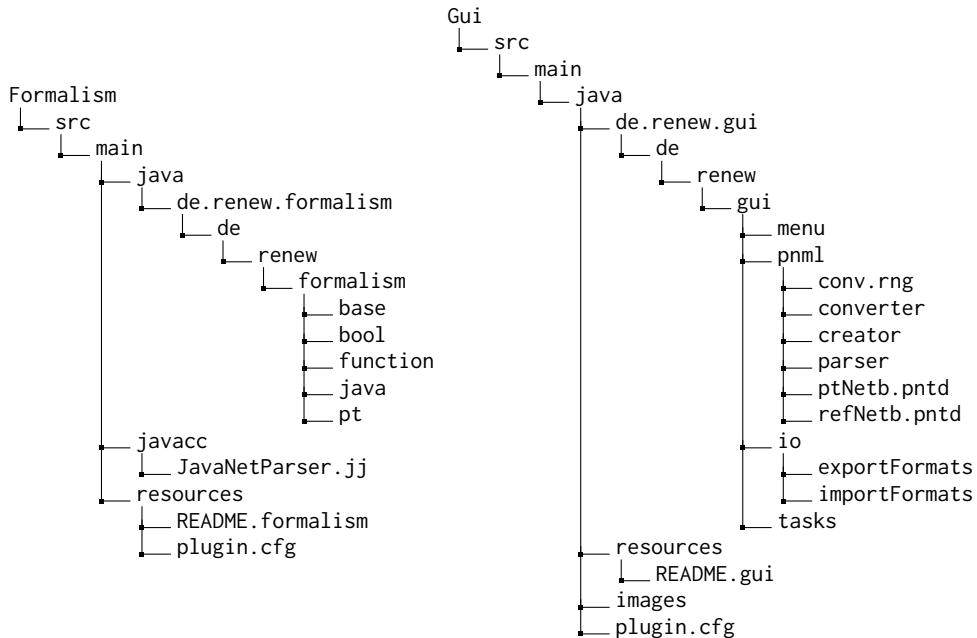


Abbildung 6.6: Resultierende Projektstrukturen

Das finale Resultat in der Abbildung ?? erlaubt, eine einfache Paketstruktur Analyse durchzuführen, um die *Split Packages* zu identifizieren.

Auf den ersten Blick kann eine Überschneidung zwischen den Gui und den RenewAnt Plugin erkannt werden, da beide den *de.renew.ant* Namensraum besetzen, der sich um bestimmte Ant spezifische Aufgaben kümmert. Aufgrund dessen wird der Namensraum in dem Gui Plugin in *task* zu den Guns-

ten des RenewAnt Plugins umbenannt. Des Weiteren könnten beide *Task's* in den RenewAnt Plugin verschoben werden, da dieser keine Abhängigkeiten in dem Gui Plugin besitzt und nicht in den Aufgabenbereich der UI fällt.

6.4.2 Gradle

Um die Zyklen zu erkennen, wird ein Gradle *build* Skript erstellt, der die Java *Source Sets* für den Kompilationsschritt definiert und die benötigten Projekte sowie Drittanbieter-Bibliotheken auf den Projektklassenpfad einbindet. Mit der Unterstützung der Gradle Projektumgebung und einer IDE können anschließend alle Abhängigkeiten aufgedeckt und auf Zyklen überprüft werden.

Für die Umsetzung der Gradle Projektumgebung wird zuerst das bestehende Ant System analysiert. Dieses besteht aus Skripten, die sich in jedem Plugin befinden und die entsprechenden Projekte ausführbare Jar's erstellen. Die Ant Skripte sind sehr ähnlich aufgebaut und wiederholen ein Erstellungsmuster für alle Plugins.

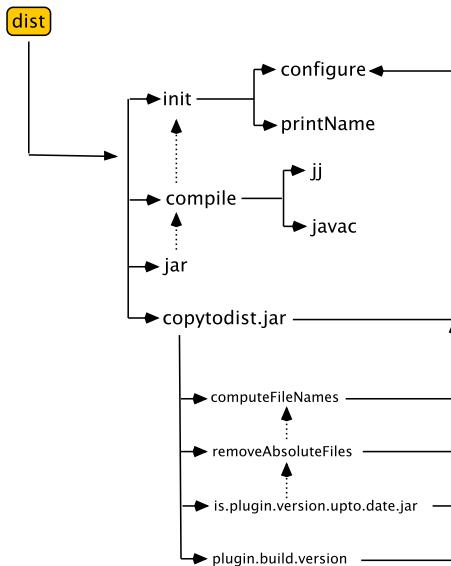


Abbildung 6.7: Ant Skript

In der Abbildung ?? ist ein Modell für die Erstellung eines Plugins mit dem Ant Werkzeug dargestellt. Ant initiiert das Skript mit Feldern und benötigten Variablen aus einer globalen Konfiguration, generiert Java Daten mit dem *jj Target*, kompiliert und verpackt die ausführbaren Klassen.

Für die Gradle Umsetzung werden Schritte, die über die etablierten Operationen zum Erstellen einer Ausführbaren Java Applikation wahrgenommen

und in der folgenden Gradle Umgebung eingebunden.
Dafür wird zuerst ein übergeordneter Gradle Projekt deklariert, das Subprojekte aus allen Plugin Verzeichnissen erstellt.

```
rootProject.name = 'renew'

file('.').eachDir { dir ->
    if (dir.name =~ "[A-Z]") include dir.name
}
```

Abbildung 6.8: Subprojekte

Die *settings.gradle* Datei in der Abbildung ??, die in der Konfigurationsphase des Gradle Lebenszyklus ausgelesen wird, ist für diesen Konfigurationsschritt zuständig und kann mithilfe von *Groovy* beliebigen Code für die Deklaration der Projekte enthalten. In diesem Fall werden alle Verzeichnisse, die mit einem Großbuchstaben anfangen als Gradle-Subprojekte eingebunden.

```
sourceSets {
    main {
        java {
            srcDirs = ["src/main/java/$moduleName", "build/generated/java"]
            java.outputDir = file("$buildDir/modules/$moduleName")
        }
    }
    test {
        java {
            srcDirs = ["src/test/java/$moduleName"]
        }
    }
}
```

Abbildung 6.9: Source Sets

Anschließend müssen die Java *Source Sets* des Projekts bestimmt werden. Um die Umsetzung so einfach wie möglich zu gestalten, wird in der *build.gradle* Konfigurationsdatei, die für die Ausführungsphase zuständig ist, eine *subprojects* Konfiguration angelegt, die für jedes Subprojekte die interne Projektstruktur definieren lässt. Diese beschreibt den Ort, an dem sich Verzeichnisse mit den Java Code und den dazugehörigen Ressourcen befinden sollen.

```
dependencies {
    automatic 'log4j:log4j:1.2.12'
    automatic 'org.freehep:freehep-graphics2d:2.4'
    implementation 'org.apache.ant:ant:1.8.2'
}
```

Abbildung 6.10: Drittanbieter-Bibliotheken

Im nächsten Schritt werden global genutzte Drittanbieter-Bibliotheken deklariert. Diese werden aus dem *Maven Repository* beim Initiieren des Projekts

geladen und auf den Klassenpfad aller Plugin Projekte eingebunden. Somit liegt die Verwaltung der Bibliotheken und der dazugehörigen Version an den *Maven Repository* und muss nicht mehr im *GitLab Repository* bereitgestellt werden.

Zusätzlich erleichtert die Deklaration der Drittanbieter-Bibliotheken, unter einer separaten Konfiguration, die Aufgabe der manuellen Erstellung der Klassenpfade und die Einbindung der Bibliotheken in der Entwicklungsumgebung, da die Konfiguration von der Entwicklungsumgebung automatisch aufgegriffen und auf das Projekt angewandt wird.

```
configurations {
    automatic.exclude module: 'hamcrest-core'
    automatic.exclude module: ':freehep-graphicsio'
    compile.extendsFrom automatic, plugin
}
```

Abbildung 6.11: Klassenpfade

Um die Klassenpfade voneinander zu trennen, werden zusätzliche Konfigurationen mit dem Namen *plugin* und *automatic* eingeführt, die Plugin Code und Drittanbieter-Bibliotheken voneinander trennen und für das Kompilieren zusammenführen. Somit können diese getrennt voneinander verwaltet, modifiziert und bei Bedarf für bestimmte Aufgaben angepasst werden.

```
jar {
    group "renew"
    baseName moduleName
    doFirst { task ->
        println "$task.project.name in progress"
    }
    from sourceSets.main.output.classesDirs
    from sourceSets.main.resources exclude("**README**")
    afterEvaluate { Project project ->
        doLast {
            println("$project.name Created: " + project.tasks.getByName('jar').archiveName)
        }
    }
}
```

Abbildung 6.12: Jar Task

Zum Schluss der globalen Konfiguration wird ein *jar* Task angelegt, der für ein gegebenes *Source Set* ein *jar*-Archiv für jedes Plugin mit den dazugehörigen Ressourcen erstellt.

Damit ist die globale Konfiguration der RENEW Plugins beendet und bereit für die individuelle Anpassung der Plugin Bedürfnisse.

Jedes einzelne Plugin benötigt einen internen Namen für die Verwaltung und die zusätzlichen Drittanbieter-Bibliotheken sowie Plugin Abhängigkeiten. Dafür wird in der Plugin *build.gradle* Konfigurationsdatei der Name

```

ext.moduleName = 'de.renew.windowmanagement'

dependencies {
    api project(":Loader")
    automatic fileTree(include: ['docking-frames*.jar'], dir: '../libs')
}
  
```

Abbildung 6.13: Individuelle Konfiguration

unter den *extension properties* deklariert und die bereits geerbten Abhängigkeiten erweitert. In der Abbildung ?? wird das WindowManagement Plugin durch eine lokale, modifizierte Drittanbieter-Bibliotheken und durch das Plugin Projekt erweitert, um alle Benötigen Abhängigkeiten abzudecken.

```

javacc() {
    configs {
        JavaNetParser {
            inputFile = file("src/main/javacc/JavaNetParser.jj")
            outputDir = file("$projectDir/build/generated/java/de/renew/formalism/java")
        }
    }
}
  
```

Abbildung 6.14: Java Compiler Compiler

Für die Konfiguration der einzelnen Plugins spielt der *JavaCC* Parser eine große Rolle, denn dieser wandelt eine Grammatikspezifikation in ausführbaren Java Code um. Die genannte Technik wird in dem *Formalism*, *Feature Structures*, *CH* und in dem *Misc* Plugin verwendet. Der *JavaCC* Parser erstellt Java Klassen, die obligatorisch für die Ausführung von RENEW sind. Dementsprechend werden diese als generierte Ressourcen im *build* Verzeichnis abgelegt und dem Java *SourceSet* beigefügt.

Jedes einzelne Plugin wird auf diese Weise konfiguriert und enthält einen Namen sowie zusätzliche Abhängigkeiten. Hiermit ist die Vorbereitung für die Modularisierung abgeschlossen.

6.4.3 Modularisierung

Nachdem alle benötigen RENEW Plugins kompiliert, verpackt und ausgeführt werden können, müssen die neu entstandenen Abhängigkeitsbeziehungen analysiert werden. Die Analyse der Plugins geschieht nun über die erstellten Gradle Scripte, die für jedes Projekt die benötigten Bibliotheken und Projekte für die Kompilation deklarieren. Aus diesen wird anschließend ein Abhängigkeitsgraph erstellt, der Zyklen und verstekte Abhängigkeiten offenlegt.

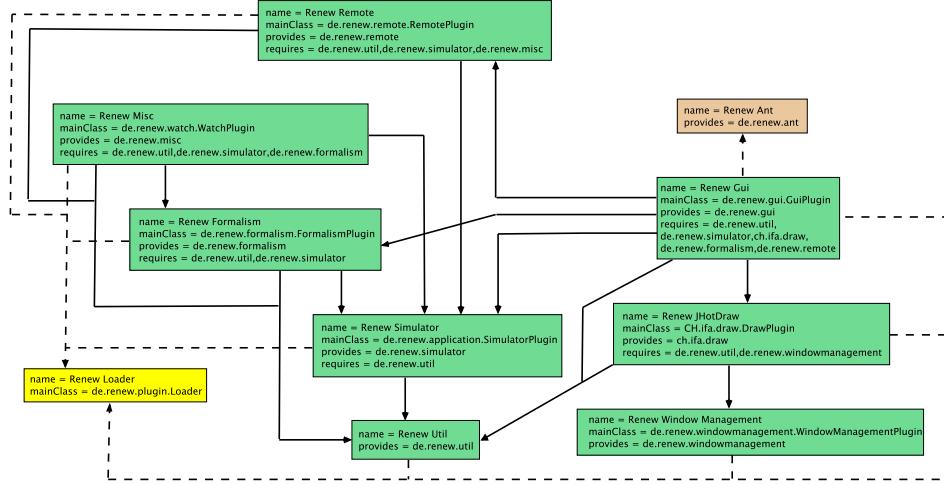


Abbildung 6.15: Kompilation Abhängigkeiten

Der neu entstandene Graf wurde im Vergleich zu dem Grafen aus der Ausgangssituation Abschnitt ?? durch ein Plugin erweitert und bindet alle Plugins an das *Loader* Projekt. Des Weiteren sind auf der Abbildung ?? keine Zyklen in der minimalen Version zu beobachten und dementsprechend müssen auch keine weiteren Anpassungen durchgeführt werden.

Die Migration von der minimalen Version von RENEW wird von den *Loader*, *Util* und *WindowManagement* Plugin eingeleitet. Diese besitzen keine Abhängigkeiten innerhalb der Plugin Menge und brauchen keinen Zugriff auf den Klassenpfad nachdem sie sich auf dem Modulpfad befinden. Im Gegensatz dazu, behalten Plugins, die sich auf dem Klassenpfad befinden und als ein unbenanntes Modul interpretiert werden, alle Zugriffsrechte auf die interne Struktur migrierter Plugins, wie bereits in den Abschnitt ?? beschrieben wurde.

Da ein Modul seine eigenen Abhängigkeiten verwalten muss, wird für jedes Plugin eine *module-info.java* Konfigurationsdatei angelegt, die alle Java internen sowie Drittanbieter Bibliotheken auflistet. Für die ersten Module werden die erforderlichen Bibliotheken inklusive dem Loader Plugin in der Konfigurationsdatei mit dem Schlüssel *requires* verankert und die dazugehörigen Drittanbieter-Bibliotheken aus dem Klassenpfad auf dem Modulpfad als automatische Module aufgesetzt.

In diesem Zustand wird RENEW kompiliert und ausgeführt. Das Ergebnis ist eine lauffähige Applikation, die ihre vollständige Funktion zugleich aus dem Modul- und Klassenpfad bezieht und identisch zu der initialen minimalen Version von RENEW funktioniert.

Im nächsten Schritt werden Plugins migriert, die nur auf die neu entstan-

```

module de.renew.loader {   module de.renew.util {
    ...
    // Java
    requires java.xml;
    requires java.desktop;
    // Third
    requires log4j;
    requires jline;
    requires commons.cli;
}
  
```

Abbildung 6.16: Benötigten Bibliotheken

nen Module aufsetzen, wie zum Beispiel das *Simulator* und das *JHotDraw* Plugin. Ihre Abhängigkeiten liegen auf dem Modulpfaden, daher gibt es keinen Grund mit dem Klassenpfad zu interagieren.

Da diese die zweite Modulschicht repräsentieren, fordern sie bestimmte Funktionalität mit dem *requires* Schlüssel aus den Loader, Util und Window-Management Plugins. Um dieser Anforderung zu entsprechen, müssen die notwendigen Plugins ihre Pakete explizit für ihre Nutzer öffnen. Dazu deklarieren die angeforderten Plugins in ihrer *module-info.java* mit dem *exports* Schlüssel Pakete, die sie für andere Plugins zur Verfügung stellen möchte.

```

module CH.ifaf.draw {
    ...
    requires de.renew.windowmanagement;
    requires de.renew.util;
    requires de.renew.loader;
    // Java
    requires java.datatransfer;
    requires java.desktop;
    // Third
    requires docking.frames.common;
    requires docking.frames.core;
    requires log4j;
    requires freehep.graphicsio;
}

module de.renew.util {
    ...
    // Interface
    exports de.renew.util;
    // Renew
    requires de.renew.loader;
    // Third
    requires log4j;
}
  
```

Abbildung 6.17: Exportierte Pakete

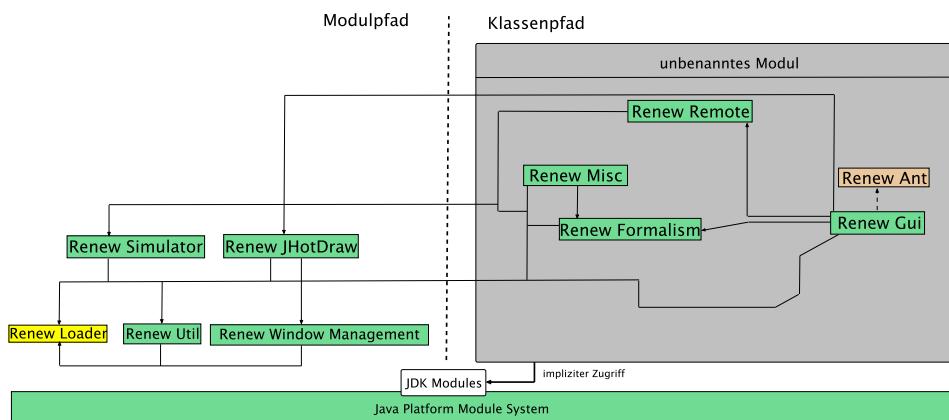
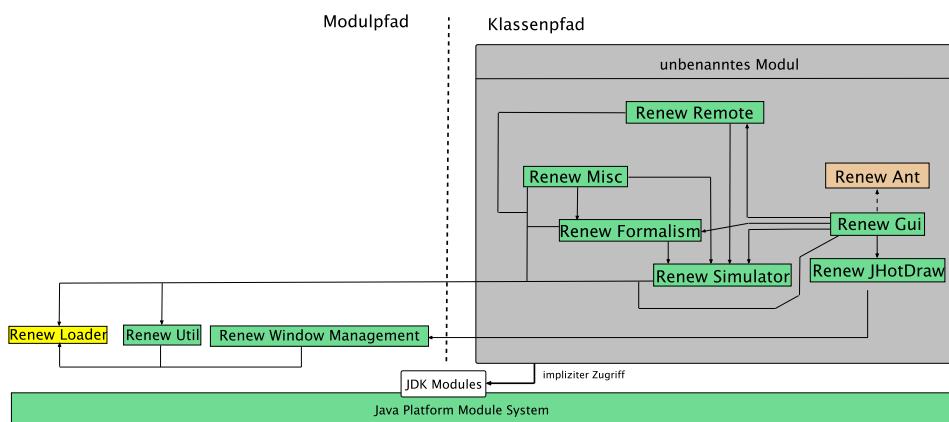
In der Abbildung ?? wurde das Util Plugin Modul angepasst und bietet jetzt das *de.renew.util* Paket für den Gebrauch an.

Die Adaption der bestehenden Module muss mit jeder neuen Modulschicht an die angeforderten Pakete und Klassen angepasst werden, bis alle Abhängigkeiten erfüllt sind.

Damit sind die notwendigen Schritte für die Modularisierung bestimmt worden und können in einem Zyklus, bis sich alle Plugins auf dem Modulpfaden befinden, durchgeführt werden. Nämlich das Auslesen und Definieren der

Plugin Kompilation- sowie Laufzeit Abhängigkeiten aus dem Gradle Build Skript mit dem Schlüssel *requires* und das Nachrüsten der Schnittstellen bestehender Module mit dem *exports* Schlüssel.

In der folgenden Abbildung wird die komplette Migration in vier Schritten dargestellt. Zuerst wird das *Loader*, das *Utils* und das *Window Management* Plugin migriert, da diese laut der Ausgangssituation keine Abhängigkeiten im Klassenpfad besitzen und nicht aus dem Modulpfad auf den Klassenpfad nicht zugreifen müssen. Im nächsten Schritt werden Plugins modularisiert, die nur auf den Modulpfad angewiesen sind. Das wären der *Simulator* sowie *JHotDraw* Plugin. Da alle Abhängigkeiten des *Formalism* und *Remote* Plugins auf den Modulpfad bewegt worden sind, können diese der Modulmenge beigefügt werden. Zum Schluss bleiben Plugins mit weit gefächerten Plugin Abhängigkeiten, wie zum Beispiel das *Misc* oder das *Gui* Plugin.



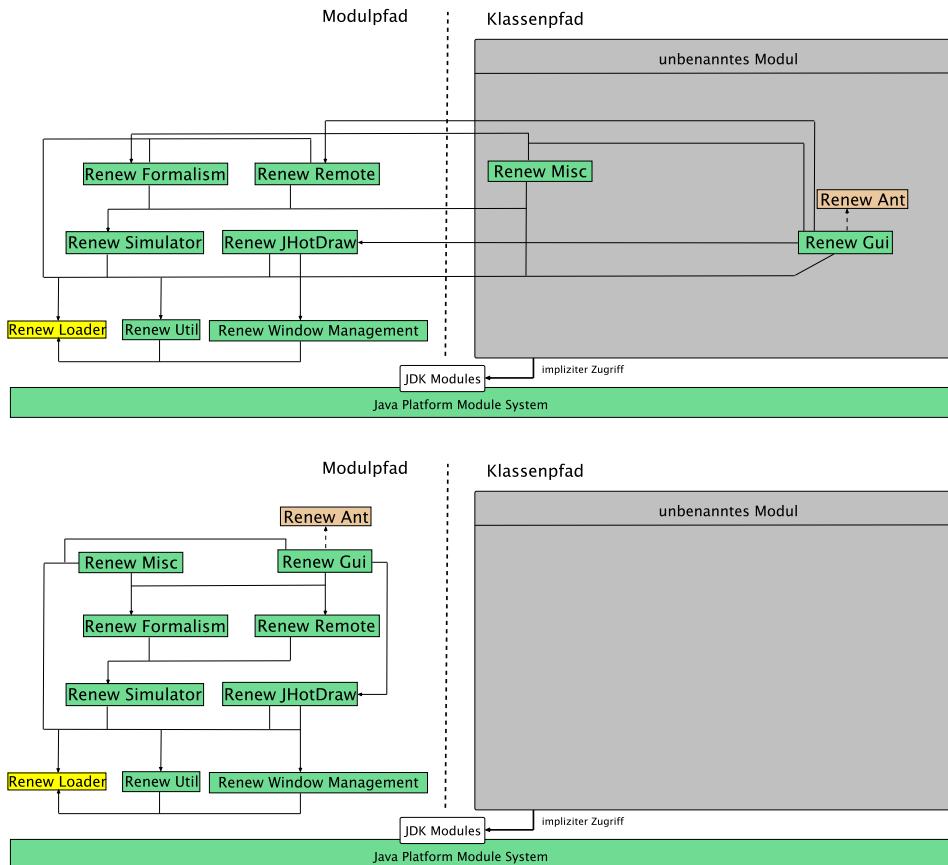


Abbildung 6.18: Migration

6.5 Evaluation

Die Evaluation des ersten Prototyps bewertet das Ergebnis des gesetzten Ziels, nämlich die Erstellung einer minimalen RENEW Version für das Modulsystem von Java. Dafür werden die Schlüsselaktionen der Migration wie Projektstruktur, Projektverwaltung und die Modulumsetzung nacheinander kritisch beleuchtet.

6.5.1 Struktur

Die Anforderung eine minimale Version von RENEW zu modularisieren und mit einem modernen Werkzeug zu verwalten wurde erfüllt. Diese Aufgabe beinhaltete das Umstrukturieren der Plugin Code Basis, die es erlaubt zusätzliche Module innerhalb eines Plugins anzulegen. Jedoch gab es unerwartete Strukturänderungen, die durchgeführt werden müssen. Zum Beispiel besitzen einige Plugins wie das Util, Loader und Simulator Plugin Test Klassen, die in die neue Struktur eingebunden werden müssen. Demzufolge musste

laut dem Maven Standard Layout ein *src/test/java/module.name* Test Resourcen, mit den dazugehörigen Test Klassen, angelegt und verwaltet werden.

Des Weiteren werden JavaCC Klassen separat von den Java Ressourcen untergebracht und von einem Gradle Plugin ausgeführt. Dies hat zur Folge, dass der generierte Code umgeleitet werden muss, um an der entsprechenden Position die Funktion zu erfüllen. Zusätzlich muss JavaCC den Java Code generieren, bevor der Java Compiler mit der Übersetzung beginnt, denn die generierten Java Klassen bilden die Basis für die Plugins und somit sind sie zwingend erforderlich für die Kompilation. Dies bezüglich wurde der *Gradle Task Graph* modifiziert, um die entsprechende Reihenfolge zu erfüllen. Daraus ergibt sich eine Komplexität, die neu für RENEW ist und in der Zukunft sorgfältig gewartet werden muss.

6.5.2 Verwaltung

Ein wesentlicher Vorteil der modernen *build* Tools wie Gradle und Maven, ist die Verwaltung der verwendeten Drittanbieter-Bibliotheken. Diese laden und binden benötigte Bibliotheken mit dem kompletten Abhängigkeitsgraphen in das Projekt ein, ohne den Zwang der Einarbeitung des Entwicklers in die Struktur der genutzten Bibliothek. Somit wird viel Zeit gespart, da man sich mit dem Kontext und den darunter liegenden Bausteinen, wie Core, Common und Util, der genutzten Bibliotheken nicht beschäftigen muss.

Die Verwaltung der Bibliotheken spielt eine große Rolle für Renew, da im Verlauf der Entwicklung, Drittanbieter-Bibliotheken angepasst wurden und somit keine Möglichkeit besteht eine saubere Version der Bibliothek einzubinden. Infolgedessen entstehen unsaubere Abhängigkeiten, die nicht aktualisiert werden können. Diese werden wie zuvor durch das Auslesen aus einem lokalen *libs* Verzeichnis in das Projekt eingebunden und müssen in der Zukunft für die saubere Umsetzung von der benutzerdefinierten Logik befreit werden.

Ein zusätzlicher Vorteil der neuen Projektverwaltung liegt an der neuen Umsetzung mit einer Programmiersprache, die es erlaubt, mit Feldern, Variablen, Schleifen und Objekten zu arbeiten. Dies hat zur Folge, dass der Übergang von den RENEW Java Code zu den *build* Skript von Gradle für den Entwickler leichter nachzuvollziehen ist und somit Akzeptanz und Anpassungsfähigkeit mit sich bringt.

Hierzu kann die Verständlichkeit des Gradle Werkzeug gegenüber dem Ant Werkzeugs an der benötigten Code Menge, die geschrieben werden muss, verglichen werden. Zum Beispiel benötigt die Ant Version von Renew, die zurzeit alle Plugins und den kompletten Umfang der Applikation verpackt,

um die 740 Zeilen für die globale Konfiguration und 135 Zeilen für jedes Projekt. Im Gegensatz dazu wiegt die minimale Gradle Version von RENEW 136 Zeilen für die globale Konfiguration und nur 20 Zeilen für jedes Plugin im Durchschnitt. Zusätzlich ist die Konfiguration von simplen Plugins mit vier Zeilen möglich und kann von jedem Entwickler erstellt und angepasst werden.

```
ext.moduleName = 'de.renew.remote'

dependencies {
    plugin project(":Simulator")
}
```

Abbildung 6.19: Remote Plugin Konfiguration

Obwohl die globale Konfiguration für die vollständige RENEW Version sich steigern wird, deuten die Zahlen auf einen geringeren Code-Abdruck des Gradle *build* Werkzeugs hin.

6.5.3 Modulumsetzung

Nachdem die Modularisierung abgeschlossen wurde, ist eine globale Sicht auf die Umsetzung möglich und offenbart Optimierungspotenzial für die Abstimmung der Modulabhängigkeiten. Zum Beispiel wird das RenewAnt Plugin nur für die Kompilation benötigt und muss nicht für die Ausführung mitgeliefert werden **??**. Daher kann dieses als eine statische Abhängigkeit im Gui Plugin verankert werden und wird der Laufzeit nicht beigefügt.

```
module de.renew.gui {
    // renew
    requires static de.renew.ant;
    //...
```

Abbildung 6.20: Statische Konfiguration

Des Weiteren wird klar, dass die RENEW Applikation in einer hierarchischen Plugin Architektur aufgebaut ist. Das heißt Plugins, die von anderen Plugins abhängig sind, erfordern das Einbinden aller darunter liegenden Schichten. Dies hat zur Folge, dass das *Util* Plugin in jedem Plugin, das auf diesen und seinen Nachfolger aufbaut, eine Deklaration benötigt.

Um die Übersicht über die unmittelbar genutzten Module zu behalten, kann mithilfe des *transitiv* Schlüssels eine angeforderte Bibliothek für alle Nutzer-Plugins offengelegt werden. Demzufolge kann ein Plugin nicht nur ein ande-

res Plugin erweitern und nutzen, sondern seinen Kontext in die Umsetzung miteinbeziehen.

```
module de.renew.misc {
    requires de.renew.formalism;
    requires de.renew.simulator;
    requires de.renew.util;      ➔     module de.renew.misc {
    requires de.renew.loader;    }           requires de.renew.formalism;
    requires log4j;
}
```

Abbildung 6.21: Transitive Konfiguration

Um den aktualisierten Java Kontext so gut wie möglich nachzubilden, unterstützt Gradle die Idee der Modularisierung und dessen Kopplungsarten, indem die Modulkopplung unter Verwendung von unterstützenden API's zum Entwerfen der transitiven und statischen Abhängigkeiten angeboten wird. Diese bestehen aus zwei weiteren Konfigurationspfaden, die durch *api* und *implementation* gekennzeichnet sind. Die *api* Konfiguration beschreibt eine transitive Abhängigkeit, die durch die Modulhierarchie weitergereicht wird und die *implementation* Konfiguration, die die genutzten Bibliotheken privat nutzt. Somit können die transitiven RENEW Abhängigkeiten, die in der *module-info.java* deklariert sind, auf die Projektstruktur mithilfe von Gradle abgebildet werden.

```
ext.moduleName = 'de.renew.misc'          ext.moduleName = 'de.renew.misc'

dependencies {                         dependencies {
    plugin project(":Formalism") ➔     plugin project(":Formalism")
    plugin project(":Simulator")       }
    plugin project(":Util")
    plugin project(":Loader")
}
```

Abbildung 6.22: Transitive Gradle Konfiguration

6.5.4 Endergebnis

Nachdem RENEW verfeinert wurde, entsteht eine eindeutige Darstellung der Plugin Abhängigkeiten und dessen Aufbau. Ersichtlich wird, dass Module eine einfache Art und Weise bieten, um den Aufbau komplexer Systeme zu verwalten und umstrukturieren.

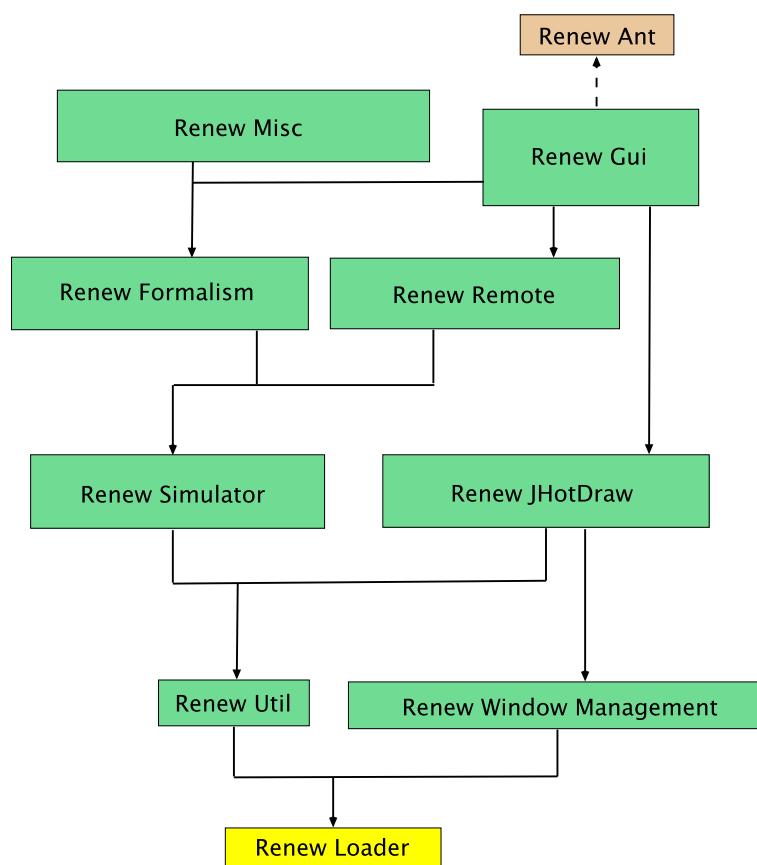


Abbildung 6.23: Minimale modulare RENEW Version

Kapitel 7

MULAN Prototyp

Die Siedler oder Settler ist ein Aufbau-Strategiespiel, das unter Verwendung der MULAN (Multi-Agent Nets) Plattform aufgebaut und mit RENEW simuliert werden kann.

Für die Gestaltung der MULAN-Settler Variation wird zunächst der Aufbau der Gesamtumsetzung geschildert und der Zusammenhang der Akteure dargestellt. Anschließend werden Anforderungen erfasst, die der RENEW Prototyp erfüllen muss, um die MULAN-Settler Variation der MULAN Plattform zu betreiben. Infolgedessen entstehen ein Implementierungsplan sowie ein Prototyp.

7.1 Anforderungen

Der MULAN-Settler RENEW Prototyp soll die MULAN Plattform unterstützen, dabei sind nur die für Settler nötigen MULAN-Plugins einzubringen, die auf dem modularen RENEW ihre Funktion erfüllen sollen. Das Ergebnis soll eine UI präsentieren und geringe Interaktionsmöglichkeiten anbieten.

Ziel des Prototyps ist die Darstellung des parallelen Betriebs von alten und neuen Softwarekomponenten, die mithilfe des Modulsystems von Java zusammengeführt werden können und in der Konsequenz als ein Beispiel für eine Migration ohne Zeitdruck und Betriebsausfall dienen.

7.2 Spezifikation

Für die Umsetzung des Prototyps muss die minimale RENEW Version neu definiert werden, denn die MULAN Plattform im Kontext des Settler Plugins benötigt eine größere Menge an RENEW Plugins. Dafür müssen die erforderliche MULAN Plugins für Settler ausgelesen werden, um anschließend die entsprechenden RENEW Abhängigkeiten abzuleiten.

Da die MULAN Plugins auf die Renew-Codebasis angewiesen sind und dessen Klassen für das Kompilieren, Verpacken und das Ausführen der Plugins benötigen, müssen die entsprechenden Referenzen in den MULAN Plugins, Ressourcen und *Ant* Skripten angepasst werden.

Die Settler und MULAN *Ant* Skripte sollen nicht auf das *Gradle* Werkzeug migriert werden, stattdessen werden lediglich Anpassungen an die neue RENEW Struktur durchgeführt.

Die Basis für diesen Prototypen sollen alle RENEW sowie MULAN Plugins aus den Abhängigkeitsgraph des Settler Spiels bilden, die aus der *plugin.cfg* Konfigurationsdatei für die Laufzeitumgebung und aus dem *Ant* Skript für die Kompiliert Umgebung abgelesen werden können.

7.3 Entwurf

Zuerst werden Komponenten in Form der benötigten Plugins aus RENEW und MULAN für das Settler Spiel erarbeitet und der Zusammenhang zwischen den Applikationen abgebildet.

Nachdem die Basis für das Settler Spiel erarbeitet wurde, müssen die *Ant Skripte* an das modulare RENEW angepasst werden, damit diese die passenden RENEW Klassen für das Kompilieren interner MULAN Strukturen genutzt werden können. Zum Teil werden es interne Plugin Klassen sein und zum Teil werden kritische Ressourcen, wie *Shadow Netze* mithilfe der RENEW Basis übersetzt. Daher ist das nahtlose Verzähnen zwischen MULAN und dem modularen RENEW Prototypen das Schlüsselkriterium der Umsetzung.

Da das modulare RENEW auf der Abschlussarbeit von Martin Wincierz [?] aufbaut, ist MULAN für die Oberflächenanpassung zu diesem Zeitpunkt nicht bereit und muss nicht nur an das modulare Renew, sondern auch an die erweiterte Oberfläche von RENEW angepasst werden.

Somit ist das Aufsetzen des Settler Spiels und der erforderlichen MULAN Plugins mit Schwierigkeiten verbunden die sorgfältig behandelt werden müssen.

7.4 Umsetzung

Zuerst wird die existierende Struktur von MULAN betrachtet und die Settler Abhängigkeit laut den *plugin.cfg*'s abgelesen. Dafür werden die Settler Abhängigkeit innerhalb RENEW und MULAN betrachtet und ein Abhängigkeitsgraph erstellt. Da die MULAN Plugins auf RENEW aufbauen, werden zusätzliche Plugins für die modulare RENEW Umsetzung durch den Abhängigkeitsgraphen aufgedeckt. Wie zum Beispiel das RENEW *Feature Structure*

Plugin, das an der Modellierung von Prozessen und der Erstellung von Ontologien beteiligt ist.

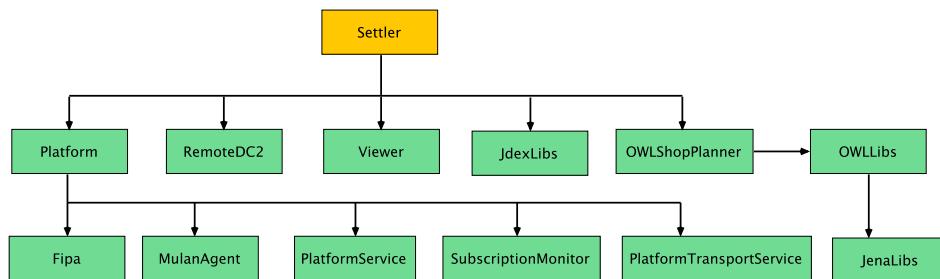


Abbildung 7.1: Settler MULAN Plugin Menge

Im Folgenden werden die benötigten RENEW Plugins modularisiert und zum ersten Mal das MULAN Settler *target* ausgeführt. Dieses soll das Settler Spiel mit allen nötigen Abhängigkeiten von MULAN sowie RENEW kompilieren und verpacken. Jedoch wird man während der Ausführung auf Plugin Klassen verwiesen, die für das Kompilieren notwendig sind und nicht als Teil des Settler *target's* sowie der *plugin.cfg's* gelistet sind.

Die Nachfolge-Analyse ergab zusätzliche Abhängigkeiten aus den *Platform Transport Service* sowie *Subscription Monitor* Plugins, die ein älteres Plugin ersetzen und noch nicht komplett in die Ant Umgebung eingebunden sind. Daher werden diese in den Settler Ant *target* verankert sowie in den *CAPA* Plugin *plugin.cfg* nachgerüstet. In der Konsequenz muss der Abhängigkeitsbaum durch die nachträglichen MULAN Plugins erweitert und auf zusätzlicher RENEW Plugins inspiert werden.

In den Abbildungen ?? sowie ?? werden die endgültigen Plugin Mengen aus MULAN sowie RENEW dargestellt, die Settler für die Ausführung benötigt.

Im nächsten Schritt müssen die Klassenpfade innerhalb der MULAN Plugins angepasst werden, denn diese verweisen auf RENEW Plugins mit der alten Projektstruktur und werden daher nicht gefunden. Um diesen Zustand zu beheben, wird die Korrektur, mithilfe der regulären Ausdrücke, MULAN übergreifend nachgezogen und enthält zum Schluss keine Referenzen auf alte sowie nichtexistierende Paketstrukturen von Renew. Die betroffenen MULAN Plugins sind: UseCaseComponents, YamlToFSConceptDiagram, KnowledgeRoundTrip, MulanDoc, FSOntologyGenerator, SLEditor, WF und das AgentRoleModeler Plugin.

Mit dem letzten Schritt wurden alle MULAN Java Klassen innerhalb der RENEW Plugin Menge richtig verbunden, dennoch bestehen Probleme bei der Referenzierung der Aufrufmethoden. Wie in der Einleitung angedeutet, setzt das modulare RENEW auf RENEW mit erweiterten Benutzeroberfläche auf und wurde bislang nicht mit MULAN zusammengeführt. Aus diesem Grund

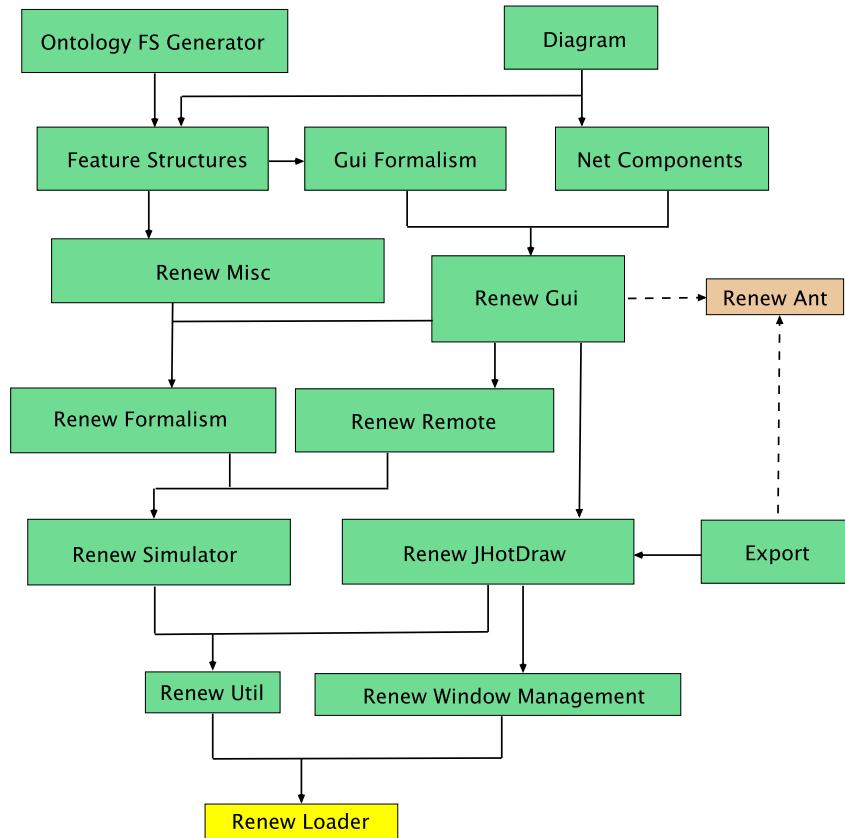


Abbildung 7.2: Minimale modulare RENEW Version für Settler

kann die *KnowledgeBaseGenerator* Klasse aus dem AgentRoleConverter Plugin nicht auf alle versprochenen Methoden von der erweiterten Benutzeroberfläche zugreifen, wie zum Beispiel das Ausführen eines Speicherdialogs. Dieses Fehlverhalten wird behoben indem Settler auf die angepasste Schnittstelle zugreift und den entsprechenden Dialog koordiniert.

Obwohl die Klassen sowie Methoden mit einander verzahnt wurden, müssen zusätzlich Ressourcen unter Verwendung des Ant Werkzeugs und speziell für RENEW implementierte *Ant Tasks* generiert werden. Diese nutzen Java Klassen, die in vielen Paketen der RENEW Plugins direkt verankert sind. Somit müssen Ant Skripte für die entsprechenden *Task's* adaptiert werden. Zum Beispiel in der *commonTasks.xml* für den *CreateShadowNetsTask*, der sich jetzt in einem anderen Paket befindet. Dieser ist zuständig für das Kompilieren der RENEW Zeichnungen in ausführbare Shadow Netze. Auch wenn der *CreateShadowNetsTask* Shadow Netze aus den RENEW Zeichnungen generieren kann, gibt es bereits kompilierte Netze die als einfache Ressourcen mit MULAN mitgeliefert werden.

Abbildung 7.3: SNS Netz

Im Gegensatz zu den RENEW Zeichnungen enthalten die Shadow Netze keine UI-Information, wie zum Beispiel Position oder Färbung der Netzelemente, jedoch enthalten diese Information über den genutzten Netz-Compiler, der für das Kompilieren der Datei zuständig war und für das Öffnen zuständig sein wird. Daher müssen bereits kompilierte RENEW Zeichnungen für die modulare RENEW Basis angepasst werden. Dafür wird mithilfe eines Text-Editors und einem regulären Ausdruck die entsprechenden Klassenverweise auf den *XFSNetCompiler* aktualisiert, um den Ausführungskontext zu entsprechen.

Das Ergebnis sowie ein Beispiel für ein aktualisiertes Shadow Netz ist in der Abbildung ?? dargestellt.

Zu diesem Zeitpunkt wurden alle Mängel und Uneindeutigkeiten zwischen Settler, MULAN und den modularen RENEW beseitigt, dennoch fehlt für die Ausführung die Zeichnung *settler.rnw*. Dieses befindet sich im Settler Plugin Projekt und wird nicht in der Standardausführung mitgeliefert, sondern bei Bedarf mit einem *sh* Skript während des Starts nachgeladen.

Da die Umsetzung auf Settler abzielt, wird dieses Skript im Ant *build* Prozess verankert und in das Settler Plugin mit kompiliert und verpackt.

Zusammengefasst ist das Settler Spiel bereit für die Ausführung und enthält alle nötigen Konfigurationen für das modulare RENEW mit der erweiterten Oberfläche. Dafür wurden zahlreiche Anpassungen durchgeführt, die eine Brücke zwischen der MULAN-Settler Variation und dem modularen RENEW bilden. Es wurden Plugins modularisiert, Ressourcen angepasst, Skripte umgeschrieben und kleine Mängel behoben.

In der Abbildung ?? ist der fertige Prototyp abgebildet, der die Spezifikation

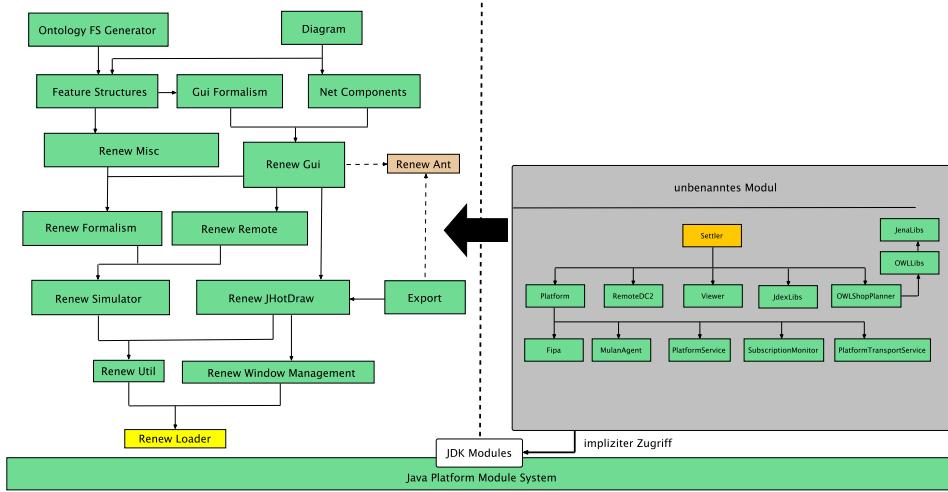


Abbildung 7.4: Settler-MULAN Variation

erfüllt und als ein praktisches Beispiel für eine Migration einer existierenden Software dient.

7.5 Evaluation

Der entstandene Prototyp deckt ein Migrationsszenario ab, das innerhalb eines Projektes mit einer großen Codebasis entstehen könnte. Die Erwartung eines nahtlosen Austausches von RENEW gegen das modulare RENEW hat sich nicht ergeben. Alle Änderungen, die in den entstehenden Prototypen resultieren, deuten auf einen mittleren Arbeitsaufwand für das initiale Aufsetzen von existierendem Alt-Code auf modulare Grundlagen. Es müssen Pfade, Ressourcen und Prozesse angepasst werden, die voraussichtlich alle Bereiche eine Applikation betreffen werden.

Nichtsdestotrotz ergaben sich auch positive Resultate. Denn, das Erweitern der existierenden Modulmenge ergab ein einfaches, übersichtliches und nachhaltiges Verfahren, welches sich in der erweiterten RENEW minimal Konfiguration erwiesen hat. Mit jedem weiteren modularisierten Plugin wurde das Hinzufügen von Plugins zu der Gesamtumsetzung immer simpler und verlangte keinen bis minimalen Aufwand für die benötigten Abhängigkeiten. Zusätzlich steigt ab einem gewissen Punkt die Unterstützung für das Kopieren der Module mit einer intelligenten IDE und bildet somit ein neues, willkommenes Hilfsmittel.

Kapitel 8

Gesamt Evaluation

Dieses Kapitel evaluierter die gesetzten Anforderungen aus der Anforderungsanalyse sowie der Anforderungsspezifikation und bewertet die Migration auf das Modulsystem von Java.

Für den Ersatz der veralteten Projektorganisation des Quellcodes wird die geforderte Projektstruktur in ein Maven Standard Layout überführt, welches den Java Code von den Java Compiler-Compiler Code trennt und die dazugehörigen Ressourcen separat verwaltet. Zusätzlich wurden die interne Struktur der Plugin Pakete, die sich mit anderen Plugins überschneiden aufgelöst und umbenannt, um den Java Anforderung des Modulpfades zu entsprechen. Darüber hinaus wurde ein Modulname eingeführt, der nach der Java Konvention mit den Organisations- oder Applikationsdomäne beginnt und mit den Feature Namen endet. Der neue Modulname fasst den Quellcode eines Moduls zusammen und ermöglicht das Erstellen zusätzlicher Module innerhalb eines Plugins, die ein gemeinsames Ziel verfolgen. Des Weiteren spiegelt der Modulname die interne Paketstruktur wieder und gestaltet die Codebasis Leserlich und Verständlich, da die Entwickler sofort ablesen können, aus welchen Modulen die Klassen stammen und für welchen Zweck diese entwickelt worden sind.

Der Aufwand für die Umstrukturierung war wie erwartet groß, denn die Modernisierung der internen Struktur eines Elements aus einem gekoppelten Verband diverse Folgen für die Zugriffe auf dessen Ressourcen und Funktionalität mit sich bringt. Demnach müssen alle Zugriffe auf das Plugin angepasst, die *import* Angaben Plugin-Weit überarbeitet und die existierende Meta-Information angeglichen werden. In Folge dessen wurden Änderungen nötig, die Plugin-Übergreifend aufgelöst werden müssen und über den Rahmen des Plugins hinausgehen. Dies führt zu Unsicherheit, da Änderungen in mehr als hundert Klassen entstehen und die Konsistenz der kompletten Applikation in Frage stellt.

Im Endeffekt, ist die Umsetzung einer neuen Projektstruktur in einer aus-

gereiften Applikation eine wichtige und verantwortungsvolle Aufgabe, die Nachhaltigkeit und Beständigkeit mit sich bringt.

Während der Modularisierung des ersten Prototyps, der die kontinuierliche Migration abdeckt, wurde der gemischte Betrieb von modularisiertem RENEW Code mit dem Alt-System betrachtet. Dementsprechend sollte die Migration nicht nur Module bilden und das Ergebnis begutachten, sondern auch die fortlaufende Integrität mit dem Altsystem inspizieren.

Die Migration verfolgte den *Chicken little* Ansatz und nutzte die von Java zur Verfügung gestellten Kommunikationsbrücke, die den Klassenpfad mit dem Modulpfad verbindet. Somit entstand kein zusätzlicher Aufwand für die manuelle Erstellung und die nachfolgende Qualitätskontrolle eines unentbehrlichen Elements. Da die Kommunikationsbrücke das Herzstück einer *Chicken littel* Migration umsetzt, ist die benannte Kommunikationsbrücke ein begehrenswertes und willkommenes Ausstattungsmerkmal des Modulsystems von Java, die einen großen Teil des Migrationsszenarios übernimmt. Dementsprechend musste im Laufe der Migration nur auf die Adaption an die neue Umgebung geachtet werden, da die Kontextübergreifende Kommunikation sowie dessen Integrität bereits von Java zur Verfügung gestellt worden ist.

Obwohl Java die Migration mit Migrationskonzepten unterstützt, entscheidet sie nicht in welchen Ablauf der Code Modularisiert werden soll. Daher wurden im Grundkapitel Konzepte vorgestellt, die nach der Größe einer Applikation Modularisierungsszenarien empfohlen. Für die RENEW Modularisierung wurde das *bottom up* Verfahren ausgewählt, da dieses als das präferierte Verfahren für gekoppelte Bibliotheksstrukturen gilt und sich gut auf das Plugin Konzept übertragen lässt. Das Verfahren modularisiert und integriert den Code, von den simplen bis zu den komplizierten Plugins, in das bestehende System und gliedert zusätzlich die Drittanbieter Bibliotheken als automatische Module ein.

Obwohl die bereits eingerichtete Projektstruktur und der Umsetzungsplan einen großen Teil der Modularisierung ausmachen, mussten die Plugins mit einer Schnittstellenbeschreibung in Form einer *module-info.java* ausgestattet werden, um als ein explizites Modul gelten zu dürfen. Das Anreichern der ersten Plugins mit einer Modulbeschreibung geschieht mühe los, da die Konfigurationsdatei eine schlichte Schnittstellenbeschreibung besitzt, die nur das nötigste anfordert und nur wenig anbietet. Jedoch mussten diese während der gesamten Migration durchgehend nach gepflegt werden, da für jedes nachträglichen Plugin, Pakete in bereits modularisierten Plugin geöffnet werden mussten. Somit entstand ein fortlaufender Arbeitsaufwand beim Überarbeiten von bereits deklarierten Schnittstellen in den benötigten Plugins.

Nichtsdestotrotz erwies sich die Durchführung als Vorbildhaft, denn die Abarbeitung der gesetzten Ziele und der entsprechenden Ausführungsschritte ergaben einen makellosen Weg vom bestehenden Altsystem auf das moder-

nen Modulsystem von Java.

Als Konsequenz der Modularisierung von RENEW samt der entsprechenden Abhangigkeiten entsteht eine Applikation mit einem globalen ubersichtlichen Abhangigkeitsgraphen, der die Evaluation der gesamten Konstruktion ermoglicht. Die Betrachtung ergab eine groe Anzahl an transitiven Abhangigkeiten, die mithilfe des *transitiv* Schlessel auf eine reduzierte Form des Abhangigkeitsgraphen runter gebrochen werden konnten und ergaben somit eine alternative Sicht auf die Plugin Konstellation. Der neue Abhangigkeitsgraph illustrierte eine naturliche Schichtenbildung von Plugins, die sich aus grundlegend bis zu erweiterten Fahigkeiten zusammensetzen. Des Weiteren impliziert die transitive Deklaration der Abhangigkeiten ihre Delegation an Module die dieses Nutzen mochten. In der Konsequenz kann der gesamte Kontext eines Plugins auf Plugins, die dieses erweitern, weitergereicht werden.

Angelwand an die RENEW Applikation, ist die transitive Deklaration der Modulabhangigkeiten eine groartige Erweiterung der Plugin Architektur, denn ab diesen Zeitpunkt wird der Kontext der zu erweiternden Plugins an den aufbauenden Plugin *automatisch* weitergereicht, ohne dass die aufbauenden Plugins sich um die globale Struktur und Abhangigkeitsverwaltung kummern mussen. Dazu gehoren die Plugin Grundlagen, Drittanbieter Bibliotheken sowie die entsprechende Versionierung.

Kapitel 9

Schluss

9.1 Zusammenfassung

Das Grundlagenkapitel behandelt das Arbeiten mit dem Klassenpfad auf der virtuellen Maschine von Java. Dazu gehören Konzepte wie der *ClassPath* und der *Klassenlader* sowie *Reflektion* und die *Interfaces*, die in der Umsetzung von RENEW eine wichtige Rolle spielen. Dementsprechend werden Mechanismen vorgestellt, wie das Auffinden der Klassenaufenthaltsorte, das sichere Laden in die Virtuelle Maschine und das Arbeiten mit unbekannten Klassentypen.

In dem Kapitel der Modularisierung wurden wichtige Konzepte und Eigenschaften der Modularisierung erarbeitet, die in der Zukunft helfen sollen, Module sauber zu entwerfen, zu erstellen oder zu bewerten. Dazu gehören kritische Modul Charakteristika, wie Modulkopplung, Modulbindung, Seiteneffektfreiheit, Modulgröße und Namensräume.

Mithilfe des Migrationskapitels werden zwei wesentliche Vorgehensweisen dargestellt, mit den eine Migration durchgeführt werden kann. Zusätzlich wird auf das Modulsystem von Java eingegangen, das eine bestimmte Vorstellung von einer Kontinuierlichen Migration auf das Modulsystem besitzt. Die Migration des RENEW Prototypen bediene sich dieser Idee und modularisiert die Plugins entsprechend dem *bottom Up* Ansatzes. Darüber hinaus wurden wesentliche Modulsystem Migrationshürden benannt, welche die essenziellen Probleme zusammenfassen.

Das Kapitel der Analyse und Ausgangssituation vermittelt die Zielsetzung und den Umfang der Abschlussarbeit sowie die nachfolgende Durchführung. Es werden Gründe für eine Migration auf das Modulsystem von Java zusammengetragen, die daraus resultierenden Konsequenzen analysiert und anschließend die Anforderung an die bevorstehenden Prototypen behandelt.

Zum Schluss wird ein aktueller Zustand der RENEW sowie MULAN Software konstruiert, die im Nachfolgenden mit einem Umsetzungsplan die gesetzte Spezifikation erreichen sollen.

In dem Kapitel des modularen RENEW Prototypen geht es um die Migration von RENEW auf das Modulsystem von Java, die eine kontinuierliche und beispielhafte Migration demonstriert. Diese beinhaltet einen Umsetzungsplan, der die Projektstruktur reorganisiert, das Gradle Werkzeug für die Organisation des Projekts einführt und anschließend das Modulsystem von Java auf die vorbereitete Code-Basis aufsetzt. Zum Schluss folgt eine Evaluation, die den Prototypen auf das Modulsystem von Java optimal abstimmt und Parallelen mit dem Gradle Werkzeug zieht.

Das Kapitel des MULAN Prototypen demonstriert mögliche Schwierigkeiten und den betreffenden Aufwand, der mit dem Austausch einer grundlegend Basis-Software in einem größeren System auftreten könnte. Da das Rahmenwerk MULAN auf dem RENEW Simulator aufsetzt und ohne diesen nicht betriebsfähig ist, ist MULAN komplett an RENEW während der Kompilation sowie der Laufzeit angewiesen. Daraus folgen Referenz- und Zugriffsschwierigkeiten, die global behoben werden müssen. Zusätzlich wird mit diesen Prototypen eine Übergangsszenario simuliert, das RENEW mit dem Gradle Werkzeug und MULAN mit dem Ant Werkzeug zugleich ein funktionierendes Ergebnis liefern.

Zum Schluss wird das Ergebnis der Abschlussarbeit evaluiert, zusammenfasst und der Ausblick für die mögliche Forschung und Ausbau der Prototypen gegeben.

9.2 Ausblick

Mit dieser Arbeit wurde das Modulsystem von Java in die RENEW Applikation integriert, organisiert und erforscht. Die Ergebnisse deuten auf eine Aufwertung der Plugin-Eigenschaften, mühelose Projektverwaltung sowie neue Möglichkeiten für die Umsetzung des Plugin-Managements. Dennoch konnten nicht alle auftretenden Fragen an Ort und Stelle geklärt werden.

Im Folgenden werden offene Fragestellungen gelistet, die für Konsistenz, Erweiterbarkeit und die Fortentwicklung von RENEW relevant sind.

Migration der Plugins

RENEW besteht aus einer großen Plugin Anzahl, die zu einem gewissen Grad mit einander verzahnt ist. Das Zusammenfügen der Plugins geschieht mit verschiedenen Verfahren, zu unterschiedlichen Zeitphasen und an diversen

Stellen. Somit ist die Modularisierung und das Erstellen der RENEW Plugin-Basis keine triviale Aufgabe und benötigt mehr Zeit als zuerst angenommen. Aus diesem Grund konnten nicht alle RENEW Plugins in das Modulsystem überführt werden und müssen in der Zukunft nachgepflegt werden.

Integration der Mulan Plugins

Der MULAN Prototyp bestätigt den parallelen Betrieb von modularen sowie nicht modularem Code-Bausteinen. Nichtsdestotrotz muss die Agentenplattform in der Zukunft auf das Modulsystem von Java migriert werden und kann von der RENEW Umsetzung profitieren.

Die MULAN Agenten Plattform orientiert sich an RENEW und baut auf demselben Plugin Konzept auf. Dies ermöglicht den Plugin Manager die RENEW sowie die MULAN Plugins zu gleich zu verwalten. Daraus folgt eine strukturelle Ähnlichkeit, die mit dem Modulsystem von Java und Gradle ausgenutzt werden kann, indem dieselbe Projektstruktur aufgesetzt und von Gradle in den RENEW Kontext für das Kompilieren mit aufgenommen wird.

Dafür muss die Projektstruktur der MULAN Plugins an die RENEW Umsetzung angeglichen und zusätzlich durch eine *build.gradle* Konfigurationsdatei erweitert werden. Somit kann Gradle die globale Konfiguration an alle MULAN Plugins delegieren und jedes MULAN Plugin durch eine individuelle Konfiguration individuell herrichten.

Abhängigkeitsmanagement

Eine wichtige mitgelieferte Fähigkeit von Gradle, ist die Abhängigkeitsverwaltung, die uns mit der passenden Bibliothek und der passenden Version aus dem Web versorgt. Die Abhängigkeitsverwaltung wurde mit den ersten Prototypen in RENEW nur zum Teil integriert, da einige Drittanbieter Bibliotheken speziell an RENEW angepasst wurden und nicht mehr aus dem globalen Maven-Repository heruntergeladen werden können. Dementsprechend werden die modifizierten Bibliotheken aus dem lokalen Verzeichnis und alle anderen aus dem Web referenziert.

Für die saubere Umsetzung der Abhängigkeitsverwaltung, müssen die modifizierten Bibliotheken von dem Arbeitsbereich gehosted werden, um das Bandbreite der Internetverbindung zu entlasten und ein schnelles Auschecken des Projekts zu ermöglichen.

Rückwärts Kompatibilität

Das Problem der *Split Packages* ist in RENEW weit verbreitet. Durch die Reorganisation der gesplitteten Pakete ändert sich der Paketname der Klassen, die an vielen Stellen im Code mit einem direkten Verwies genutzt werden. Obwohl die Klassen Verweise gepflegt wurden, können bereits erstellte oder generierte Ressourcen auf der lokalen Maschine des Nutzers, wie zum Beispiel

die RENEW Netze Zeichnungen, nicht korrekt angezeigt werden, da diese auf den veralteten voll-qualifizierten Klassennamen referenzieren.

Für die valide Darstellung der *legacy* Zeichnungen, benötigt RENEW eine Transferfunktion, die Ressourcen nach funktionsuntüchtigen Klassen abtastet und diese vor dem Öffnen substituiert.

Migration der Ant Tasks

Die Umsetzung des BPMN Plugins beinhaltet die Aufnahme der benötigten Ant Ausführungsschritte in die neue Gradle Build-Umgebung, die Klassen und Ressourcen für den Betrieb generieren. Dank der nahtlosen Integration von Ant in Gradle, war es möglich die Ausführungsschritte mit minimalem Aufwand aus Gradle aufzurufen und die Funktionalität nahtlos in die Produktion der Bibliothek einzubinden. Jedoch wird der Aufruf über eine von Gradle zur Verfügung gestellte Ant Instanz durchgeführt, welche die Arbeitsweise von Ant voraussetzt. Somit muss *Renew* Entwickler zwei Disziplinen beherrschen, die sich im selben Kontext befinden.

Die Ant Ausführungsschritte können im *Groovy*, *Java* oder *Kotlin* Format als *Gradle Tasks* umgesetzt werden, um die Prozedur ohne Einschränkungen, zusätzlichen Aufwand und Ant Kenntnisse betreiben zu können.

Zerlegung der Plugins

Dank der Injektion der Moduleigenschaften in Plugins, können die Plugins in das Modulsystem von Java aufgenommen werden und verfügen nun über erweiterte Services, die das Verwalten und Validieren der Code-Bausteine automatisch übernehmen. In der Konsequenz ermöglichen die eingeführten Änderungen den Betrieb der Plugins auf dem Modulpfad, die nun auf alle neuen Features des Modulsystems zugreifen können. Dennoch können Plugins ihre Moduleigenschaften weiter schärfen, um die Idee der Modularisierung weiter zu verfolgen.

Einer der zentralen Eigenschaften der Module ist ihre Größe, die einen klaren übersichtlichen Aufgabenbereich abdeckt und leicht verstanden sowie ausgetauscht werden kann. Diese Eigenschaft wird nicht von jedem Plugin-Modul umgesetzt, da bestimmte Plugins komplexe Aufgaben lösen und die komplette Verarbeitungskette intern umsetzen. Somit führt die Weiterentwicklung der Plugins zu großen und übersichtlichen Komponenten, die komplexe Aufgaben lösen und die Verarbeitungskette der Teilaufgaben unübersichtlich arrangieren, sodass die Logik nur schwer zu überblickbar ist. Denn, nicht jedes Detail kann sofort wahrgenommen und in der globalen Umsetzung richtig interpretiert werden.

Um die Verarbeitungsschritte in einem größeren Plugin voneinander zu differenzieren, müssen Plugins, die bereits als ein ganzes einzelnes Modul umgesetzt worden sind, in kleinere Aufgabenbereiche aufgeteilt und in Modu-

leinheiten zerlegt werden. Infolgedessen entstehen gekapselte Verarbeitungsschritte die separat verstanden sowie ausgetauscht werden können und bringen damit *Renew* ein Schritt näher zu einer erstrebenswerten Perspektive der perfekten Umsetzung eines Modularen Systems.

Die Aufteilung der großen Plugins in kleinere Moduleinheiten, ist eine individuelle und zeitintensive Aufgabe, die ein tiefgründiges und theoretisches Verständnis des Plugins erfordert. Denn, nicht jede mögliche Aufteilung der Code-Blöcke garantiert die Korrektheit der Plugins in ihrer neuen Form.

In der Konsequenz konnte diese Abschlussarbeit die Aufgabe nicht umsetzen. Trotzdem bleibt die Aufteilung der Plugins in kleine beherrschbare Module eine grundlegende Herausforderung, die in der Zukunft thematisiert werden muss.

Grundriss Prototyp

Die momentane Umsetzung von RENEW basiert auf einem Klassenlader System, welches das Entladend der Plugin nicht erlaubt. In der Konsequenz bleiben die Klassen auf dem Klassenpfad, sind immer erreichbar und können nicht aktualisiert werden. Für das geschilderte Problem wurde ein Lösungsansatz ?? vorgestellt, welcher auf dem Modulsystem von Java aufbaut und eine mögliche Umsetzung für den dynamischen Lademechanismus abbildet. Der vorgestellte Grundriss demonstriert wie die Plugins separat über Modulschichten verteilt und verwaltet werden können.

Für die Integration des Grundrisses in das Plugin Management, muss der Plugin Manager eine neue Aufgabe übernehmen, nämlich die Aufteilung der Plugins auf die Modulschichten mit den dazugehörigen Drittanbieter Bibliotheken, die Verknüpfung der Modulschichten untereinander und das Laden und Entladen von bestehender oder neu hinzugekommener Plugin Schichten. Des Weiteren müssen für die UI optionale Plugins mit dem *Service Loader* von Java erstellt und verknüpft werden, sodass die UI für sie unbekannte Plugins manipulieren kann.

Die Aufgabe beschäftigt sich mit der Umsetzung lang gewünschten Verhalten, welches nun mit Java ohne zusätzlichen Drittanbieter Rahmenwerks angegangen werden kann.

