

ABSCHLUSSARBEIT

MODULARISIERUNG DES RENEW
PLUGIN SYSTEMS

ARKADI DASCHKEWITSCH

19. MAI 2019



UNIVERSITÄT HAMBURG
DEPARTMENT OF COMPUTER SCIENCE
CHAIR OF THEORETICAL FOUNDATIONS OF COMPUTER
SCIENCE

BETREUT DURCH DANIEL MOLD

Kurzfassung

Diese Arbeit behandelt die Modularisierung und die Erweiterung des Petri-
netz Simulators *Renew* auf eine Mikroservice Architektur.

Renew wurde für die Modellierung von komplexen und verteilten Systemen mit Hilfe der Petrinetze entwickelt. Da die Software viel Fachwissen voraussetzt und sensibel auf Änderungen ungeübter Entwickler reagierte, entstand eine Plugin-Architektur, die *Renew* mit Funktionalität anreichern ließ ohne ihre innere Struktur zu verletzen.

Diese Architektur ermöglichte die Entwicklung einer Großzahl an Plugins von Mitarbeitern aus verschiedenen Forschungsbereichen. Dementsprechend blieb die Plugin-Architektur von *Renew* in den letzten sechzehn Jahren unverändert.

Obwohl die Plugin-Architektur sich langfristig bewährte, ist sie nicht für die modernen Herausforderungen gewappnet. Da die moderne Rechenleistung sich in der parallelen und verteilten Datenverarbeitung befindet, kann *Renew* in jetzigem Zustand sich diese nicht zunutze machen. Darüber hinaus verpflichtet das neu eingeführte Modulsystem von Java die Aufteilung existierender Systeme auf entkoppelte, eigenständige Einheiten. Ziel dieser Arbeit ist die Vorbereitung und Umstrukturierung von *Renew* für die Mikroservice-Architektur, um ihre interne Struktur verteilt auszuführen.

Es existieren bereits Arbeiten im *Renew* Kontext, die nach einem verteilten Unifikationsverfahren streben und sich mit möglichen Optimierungsabläufen befassen. Diese konzentrieren sich auf die verteilte Umsetzung des Algorithmus und die Schwächen in der bestehenden Implementation. Dennoch bleibt die Grundstruktur identisch.

Im ersten Abschnitt dieser Arbeit wird die Code Basis um-strukturiert und gestaltet *Renew* modular. Folgerichtig wird *Renew* auf die Korrektheit überprüft und nicht mehr unterstützten Legacy Architektur Entscheidungen überarbeitet. Im zweiten Abschnitt wird *Renew* als ein Mikroservice Verband aufgesetzt.

Inhaltsverzeichnis

Kurzfassung	I
Tabellenverzeichnis	V
1 Einleitung	1
1.1 Kontext der Petrinetze	1
1.2 Ein Petrinetz Simulator	2
1.3 Konstellation des Simulators	2
1.4 Ziel	3
1.5 Umsetzung	3
1.6 Aufbau der Arbeit	4
2 Grundlagen	5
2.1 Java Virtual Machine	5
2.2 Klassenpfads	6
2.3 Classloader	8
2.4 Schnittellen	12
2.5 Reflection	13
3 Modularisierung	18
3.1 Ziele der Modularisierung	19
3.2 Modulstruktur	20
3.3 Moduleigenschaften	22
3.4 Modulentwurfskriterien	22
3.5 Modularten	24
3.6 Modulkopplung	27
3.7 Module-Classloading	29
4 Migration	33

4.1	Legacy-System	34
4.2	Migration	35
4.3	Migrationshürden	35
4.4	Migrationsarten	36
5	Analyse der Ausgangssituation	42
5.1	Motivation	43
5.2	Ausgangssituation	47
5.3	Auswirkung	48
6	Prototypen	52
6.1	Anforderungen	52
6.2	Spezifikation	54
6.3	Entwurf	54
6.4	Umsetzung	57
6.5	Evaluation	66

Abbildungsverzeichnis

2.1	Java Classpath	6
2.2	Jar Classpath	7
2.3	Classloader System	8
2.4	Klassensuche	10
2.5	Namensräume	11
2.6	Schnittstelle	12
2.7	Aufruf einer Methode	15
3.1	Simple Modulstruktur	20
3.2	Schematischer Aufbau eines Moduls	21
3.3	Modulbindung und Modulkopplung	24
3.4	Modularten	25
3.5	Modulzugriffsrechte	26
3.6	Die Schnittstellenbeschreibung <i>module-info.java</i>	28

3.7	Abwandlung der Kopplungsarten	29
3.8	Modul Classloading	30
3.9	Modularisierte rt.jar Bibliotheken	31
4.1	Plattform Migration	38
4.2	<i>Top Down</i> Migration	39
4.3	<i>Bottom Up</i> Migration	40
5.1	Gui Plugin-Abhangigkeiten	47
6.1	Projektstruktur	55
6.2	Gradle Konfiguration	55
6.3	Modulumwandlung	56
6.4	Gui Projekt	57
6.5	Formalism Projekt	58
6.6	Resultierende Projektstrukturen	58
6.7	Subprojekte	59
6.8	Source Sets	60
6.9	Drittanbieter-Bibliotheken	60
6.10	Klassenpfade	61
6.11	Jar Task	61
6.12	Individuelle Konfiguration	61
6.13	Kompilation Abhangigkeiten	62
6.14	Module Infos	63
6.15	Module Infos	64
6.16	Migration	65
6.17	Remote Plugin Konfiguration	67
6.18	Statische Konfiguration	68
6.19	Transitive Konfiguration	68
6.20	Transitive Gradle Konfiguration	69
6.21	Minimale modulare Renew Version	70

Tabellenverzeichnis

Kapitel 1

Einleitung

1.1 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 gefärbten 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 des 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 werden und zwingen die mit einander verbundenen 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 anspruchsvolle Modellierungswerkzeug bietet, bleibt das gesamte Netzwerk statisch. Demzufolge wurde der nächste Evolutionsschritt in der Entwicklung der Petrinetze mit den Referenznetz Formalismus gemacht. 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.

1.2 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 umstrukturiert. Ab diesen Punkt kann Renew über die Plugin Schnittstellen erweitert werden, ohne die existierende Logik zu beeinflussen.

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 mit Hilfe der Plugins modifizieren. Der Plugin-Manager baut auf zwei primären Namensräume auf. Zum einen braucht dieser zusätzliche Bibliotheken zum verwalten seiner Umgebung und zum anderen braucht er Plugins, die Funktionalität mit sich bringen.

1.3 Konstellation des Simulators

Mit der Plugin Architektur hat Renew ihren Lebenszyklus weit überschritten, denn der Plugin-Manager und die Kernfunktionalität blieb lang unverändert. An manchen Stellen datiert die Code Basis aus dem Jahr 2002 (JDK 1.4). Somit entsprechen die erstmaligen Gestaltungsmöglichkeiten, Architekturentscheidungen und ihre Umsetzung, nicht mehr den aktuellen Stand der Technik (JDK 12). Vor allem durch die Einführung des Modulsystem von Java, mit dem der JDK sowie der darauf aufbauenden Code modularisiert wird. Im Zuge dessen ist das Portieren der Applikation nicht mängelfrei. 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 zusätzlichen Plugin-Code eine zentrale Funktion im System vertretert.

Indem die Applikation neu strukturiert wird, sind zusätzliche Mängel zu erwarten, denn das Vernachlässigen der Gesamtarchitektur und die Konzentration auf eigne Plugins führt zu Code-Duplizierung und Problemen in der Verständlichkeit der Plugin-Abhängigkeiten.

Beifolgend stellt sich die Frage: Wie Portierbar ist Renew und was muss getan werden, damit der Umstieg auf das Modulsystem von Java gelingt.

1.4 Ziel

Das Ziel dieser Arbeit ist die Anpassung von Renew an die neuen Anforderungen der Java Laufzeitumgebung sowie das Ausarbeiten einer möglichen Architektur für die verteilte Ausführung. Dementsprechend soll Renew einen einfachen Einstieg in die existierende Kernlogik geben, sowie die Entwickler-Fähigkeiten in der Forschung verteilter Systeme vorantreiben, indem selbst erstellte Simulation verteilt ausgeführt werden dürfen.

1.5 Umsetzung

Als Teil der Umsetzung entsteht ein Prototyp, der ein Teil von Renew modular restrukturiert. Demzufolge wird die Code-Base sowie Design-Entscheidungen modernisiert und auf die Java 11 Version angepasst. Dabei soll der Schwerpunkt dieser Arbeit beim Plugin-Manager liegen, der die zentrale Aufgabe der Plugin-Verwaltung trägt. Um das modulare Renew ausführen zu können, sollte die Software zusätzlich mit Hilfe eines modernen Build-Management-Systems compiliert und ausführbar verpackt werden. Im nächsten Schritt werden die Plugins für die verteilte und kooperative Arbeiten vorbereitet. Dementsprechend wird eine Architektur benötigt, die den modularen Prototypen erweitert. Somit entsteht eine Mikroservices-Architektur, die von einander entkoppelte Einheiten ein globales Ziele verfolgen lässt.

Für die Implementation werde ich verteilte Software-Umsetzungen untersuchen und die Vor- und Nachteile ermitteln. Infolgedessen wird mittels der entstehend Rezension die Umsetzungsentscheidung der verteilten Architektur getroffen.

1.6 Aufbau der Arbeit

Hier kommt der Aufbau

Kapitel 2

Grundlagen

2.1 Java Virtual Machine

Gemessen am Interesse der Anwender und an seiner 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 besteht. Dementsprechend ist Java im Laufe der Zeit durch Designmustern, Architekturkonzepten, Paradigmen und aktuellen Sicherheits- sowie Industriestandards erweitert worden. Da Renew mit Hilfe 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 Plugins eine große Rolle für die entstandene Architektur. 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*, *Class-loaders* und *Reflection* erläutert mit Hilfe dessen die Renew Plugin-Architektur umgesetzt wurde.

2.2 Klassenpfads

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

Um die compilierten Klassen zu laden, wird von der virtuellen Maschine Ortsangaben mit entsprechenden Code erwartet. Die Ortsangaben nennt man *Classapth* 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 *Classlaoder System* die gewünschten Klassen erfassen und in die virtuelle Maschine laden.

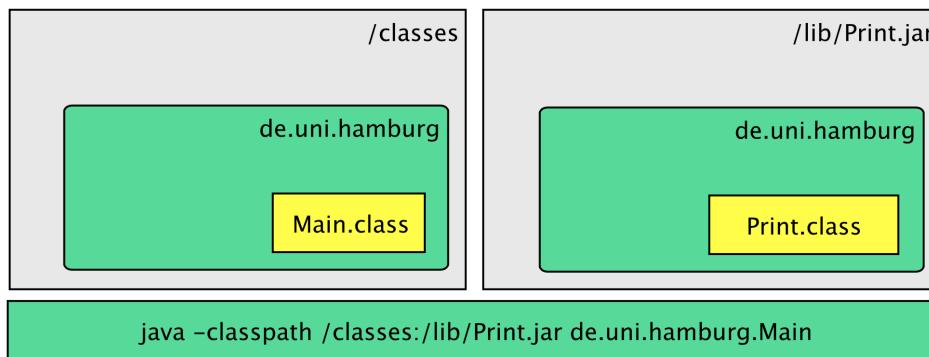


Abbildung 2.1: Java Classpath

In Beispiel 2.1 besteht der Klassenpfad aus einem Ordner 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, auf dass der *Applikation Classloader* diese durchsuchen kann. Abschließend braucht Java einen Startpunkt, mit dem die Applikation ihre Ausführung beginnt.

Beim Starten der Applikation werden Klasse instanziert, indem der Klassenpfad von Links nach Rechts nach dem benötigten Typ durchsucht und erstellt. Somit hat der Klassenpfad eine interne Ordnung und eine Abarbeitungsreihenfolge.

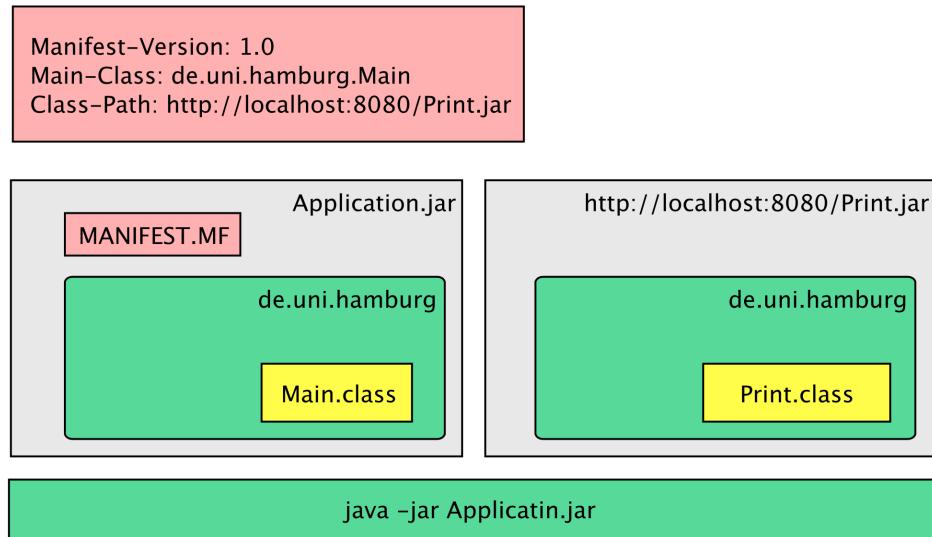


Abbildung 2.2: Jar Classpath

Im Beispiel 2.1 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.

Mit Hilfe der 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 2.2 der Klassenpfad durch ein Archive 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 mitgelieferten Meta-Information komplett ist.

2.3 Classloader

In den vorherigen Beispielen [2.1, 2.2] wurde die Bedeutung und die Rolle des Klassenpfads für die Applikation beschrieben, dennoch muss dieser zuerst verarbeitet werden. Diese Aufgabe wird von dem Classloader übernommen, der eine zentrale Rolle in jeder Applikation spielt, zumal er nach benötigten Java Klassen für die Instantiierung der entsprechenden Typen sucht. Da es eine wichtige Aufgabe ist, wird die Verantwortung für das Laden der Klassen über eine Menge von Classloader aus dem *Classloader System* aufgeteilt.

2.3.1 Classloader System

Das *Classloader System* besteht aus drei integrierten Classloader, 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.

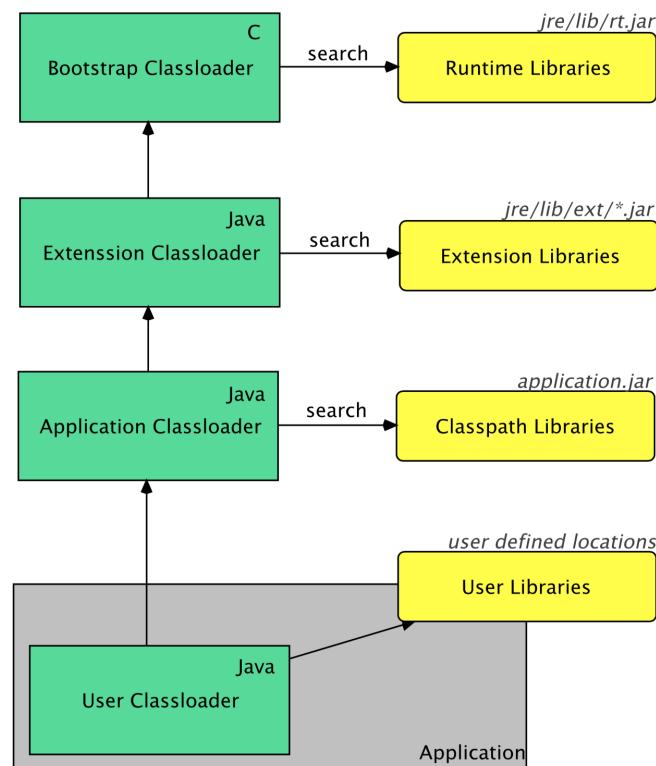


Abbildung 2.3: Classloader System

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

Das in 2.3 abgebildete Classloader System verhindert, dass Code aus weniger sicheren Quellen vertrauenswürdige Core-API-Klassen ersetzt, indem der selbe Name als Teil der Core-API angenommen wird. Daraus folgt ein Delegierungsmodell, welcher eindeutige Klassen garantiert, da die Klassensuche von Oben nach Unten der Classloader-Hierarchie abgearbeitet wird.

2.3.2 Delegierungsmodell

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

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

Wenn während der Suche ein übergeordneter Knoten eine bestimmte Klasse findet, dann wird diese Klasse die Baumhierarchie herunter zu der Anfra-

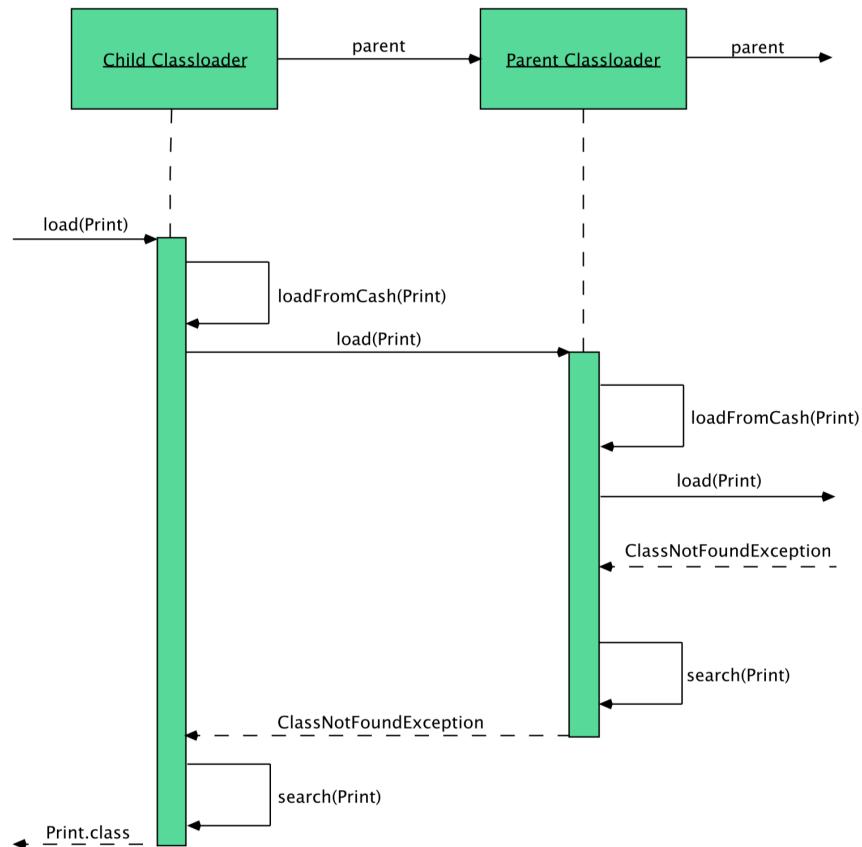


Abbildung 2.4: Klassensuche

ge delegiert. Andernfalls versucht der zuständige Classloader als letzter die Klasse selbstständig zu laden. Dies bedeutet, dass eine Klasse normalerweise nicht nur in dem Classloader sichtbar ist, der sie geladen hat, sondern auch für alle untergeordneten Instanzen. Dies bedeutet auch, wenn eine Klasse von mehr als einem Classloader in einem Baum geladen werden kann, wird immer die Klasse des übergeordneten Classloader eingelesen. Dennoch wird vor jedem Laden der Klasse der Cash-Speicher des Classloaders nach der gewünschten Instanz durchsucht. Wenn diese existiert, wurde die Suche bereits zuvor durchgeführt und keiner der übergeordneten Classloader außer dem jetzigen, war fähig die Anfrage zu beantworten. Somit kann die Suche beschleunigt werden, indem der Type sofort zurückgegeben wird.

2.3.3 Namensräume

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

Ein *Namensraum* ist eine Gruppe von Klassennamen, die von einem bestimmten Classloader geladen worden ist. Wenn ein Eintrag für eine Klasse einem *Namensraum* hinzugefügt wurde, ist es nicht möglich, eine andere Klasse mit dem selben 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 Classloader mit dem separaten *Namensraum* erstellt wird.

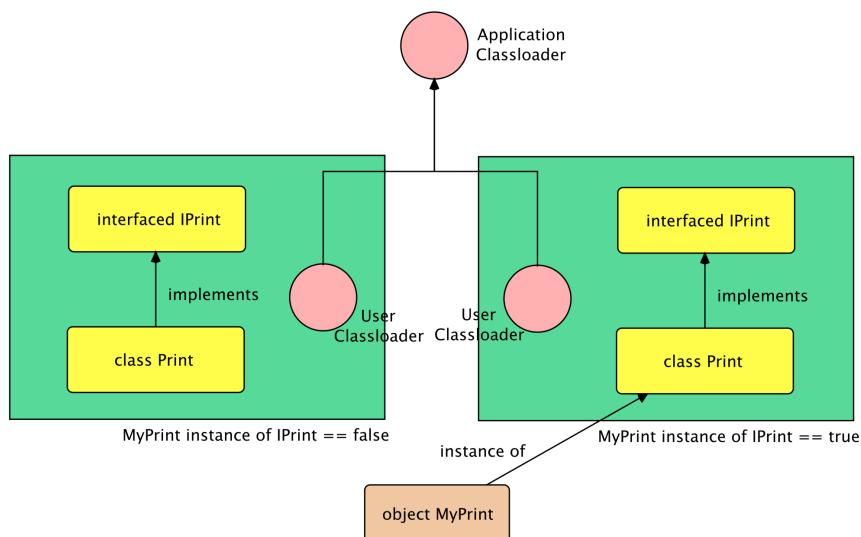


Abbildung 2.5: Namensräume

Die Abbildung 2.5 zeigt ein Beispiel für eine Klassenidentitätskrise, die sich ergibt, wenn eine Schnittstelle und die zugehörige Implementierung jeweils von zwei separaten Classloader geladen werden. Obwohl die Namen und binären Implementierungen der Schnittstellen und Klassen gleich sind, kann eine Instanz der Klasse von einem Classloader nicht als Implementierung des Interfaces von dem anderen Classloader erkannt werden. Bei Wunsch kann dieser Umstand gelöst werden, indem das Interface eine Ebene höher rutscht und von den Application Classloader geladen wird. Somit implementieren

beide *Print* Klassen die selbe Schnittstelle.

Der Klassen Namensraum bieten zusätzliche Sicherheitsfunktionen wie die Kapselung privat deklarierter Pakete. Denn die Namensräume verhindern, dass weniger vertrauenswürdiger Code, der aus der Applikation oder benutzerdefinierte Classloader geladen worden ist, direkt mit mehr vertrauenswürdigen Klassen interagieren kann. Beispielsweise wird die Kern-API vom *Bootstrap-Classloader* geladen, diese kann *package private* Code enthalten, der bei Anfrage nicht an die unterliegende Classloader weitergereicht wird. Auch wenn ein untergeordneter Classloader die Paketstruktur der Core-API nachahmt, wird diese nicht als Teil der Java Core-API anerkannt, da dieser von den falschen Classloader 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.

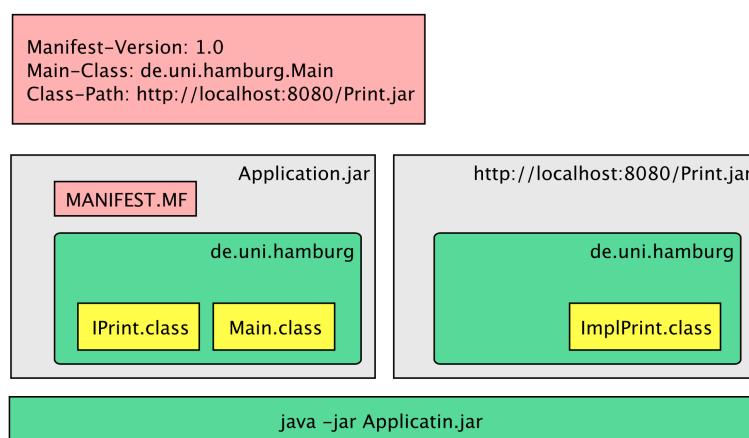


Abbildung 2.6: Schnittstelle

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

Im Beispiel 2.6 wird der Vorteil des Schnittstellenvertrags demonstriert, der das Ausführen, für die Applikation unbekannte *PrintImpl* Umsetzung, durch eine einfache Schnittstellenbeschreibung *IPrint* garantiert. Solange die Implementation sich auf der selben Klassenpfadhierarchie befindet wie die Schnittstelle, wird diese während der Laufzeit auf Kompatibilität geprüft und angewandt. Somit kann dynamische Klassenbindungen während der Laufzeit entstehen und Laufzeitbibliotheken ausgetauscht werden ohne die Applikation zu verändern. Hätte man die Schnittstelle nicht genutzt, würde man die Implementation nur als ein Objekt Type instantiiieren 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 Reflection

Reflection 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, mit Hilfe der Reflection-API während der Laufzeit instanziert, bearbeitet und genutzt werden. Somit ermöglicht Reflection das Arbeiten mit Klassen von den 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 seinen Applika-

tion von andren Nutzern und Entwicklern erweitern lassen, ohne das diese bei jeder Änderungen die komplette Applikation umbauen. Somit stellt sich die Frage, wie erstellt man ein Mechanismus der mit beliebigen Klassen arbeiten kann. Man könnte mit dem 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 Kern und Drittanbieter Bibliotheken über die wir keine Kontrolle verfügen. Somit ist die Erweiterung der gesamten Codebasis mit der entsprechenden Schnittstelle oder eine Verschachtlung von *instanceof* Blöcken keine simple oder saubere Lösung.

Dementsprechend sollte Reflection 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.

In der Abbildung 2.7 wird ein Ablauf eines Methodenaufrufs mit Hilfe von Reflection visualisiert. Im ersten Schritt muss die Methode gefunden werden. Da wir den Typen nicht kennen und nur des Objekt Typs deklarieren Methoden nutzen können, lassen wir das Objekt sich selbst inspizieren 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.

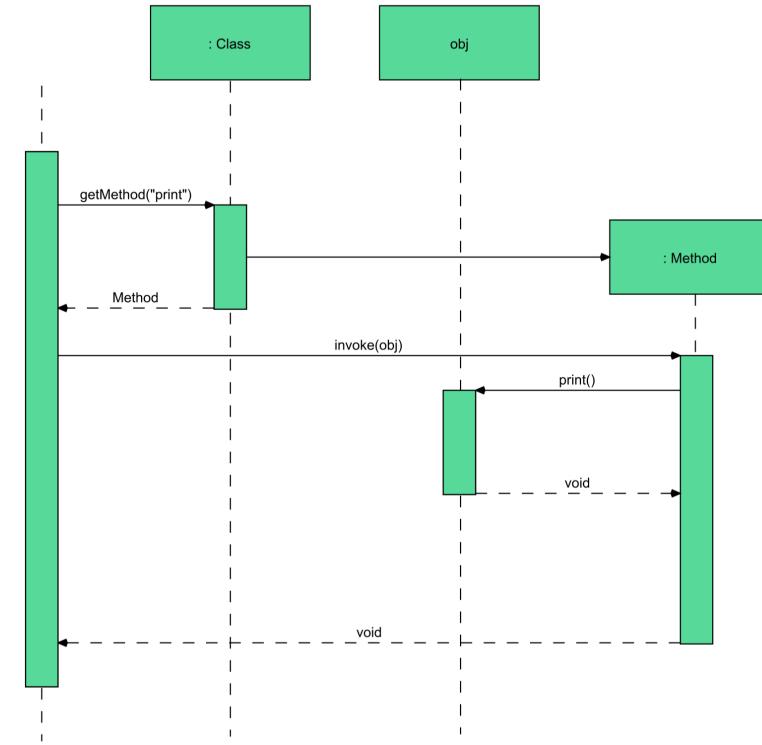


Abbildung 2.7: Aufruf einer Methode

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

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

Mit Hilfe der Klassen Objektes eines Typs können die oben genannten Objekte erzeugt und manipuliert werden. Diese bietet eine Schnittstelle für das Abfragen ihrer Struktur an den Nutzer und liefert ein Objekt aus dem `java.lang.reflection` Paket zurück.

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

- Constructor[] *.class.getConstructors();

Um den Zusammenhang und den Nutzen von Reflection darzustellen wird in der Abbildung 2.1 ein Szenario durchgespielt, das eine unbekannten Typen mit Hilfe 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 mit Hilfe von Reflection 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     // Instantiierung
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: Reflection in Aktion

Wie in der Abbildung 2.1 dargestellt ist Reflection 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* mit Hilfe von Reflection realisiert, indem ein Framework, wie zum Beispiel Spring, die entsprechenden Implementierungen für ein Interface sucht und initiiert. In Diesem Zusammenhang wird Anhand des *implement* Schlüssels und zusätz-

licher 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 entsprechenden Module durch das Analysieren der *web.xml* und Anfordern der passenden URI.
- JUnit verwendet Reflection, um die Methoden einer Klasse nach Test-Annotation zu durchsuchen, um diesen anschließend aufzurufen.

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 der lose gekoppelten Systeme verfolgt. Es generiert Struktur in großen Softwareprojekten, indem das Gesamtprodukt in kleine, praktische Bestandteile zerlegt wird. Die Entwicklung von kleinen Projekten mit übersichtlicher Codebasis ist einfach zu überblicken und braucht keine strukturelle Basis, 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 Projekt wächst auch 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 von einander 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 eine Aufstellung von Zielen und Qualitäten den 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 internen 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 System 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 bleiben, auch dann wenn Teile des System nicht reagieren.

3.2 Modulstruktur

Die zuvor aufgeführten Ziele der Modularisierung liefern bereits eine Idee davon, was für Anforderungen Module erfüllen müssen, um von einem Modul sprechen zu können. Primär erfüllt ein Modul einen abgeschlossenen Aufga-

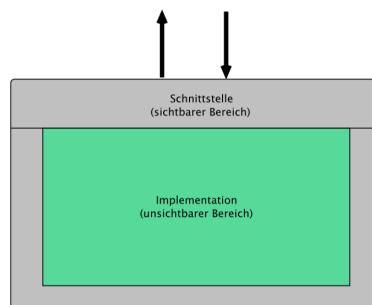


Abbildung 3.1: Simple Modulstruktur

benbereich 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 Gesamtimplementation auf die Schnittstellen.

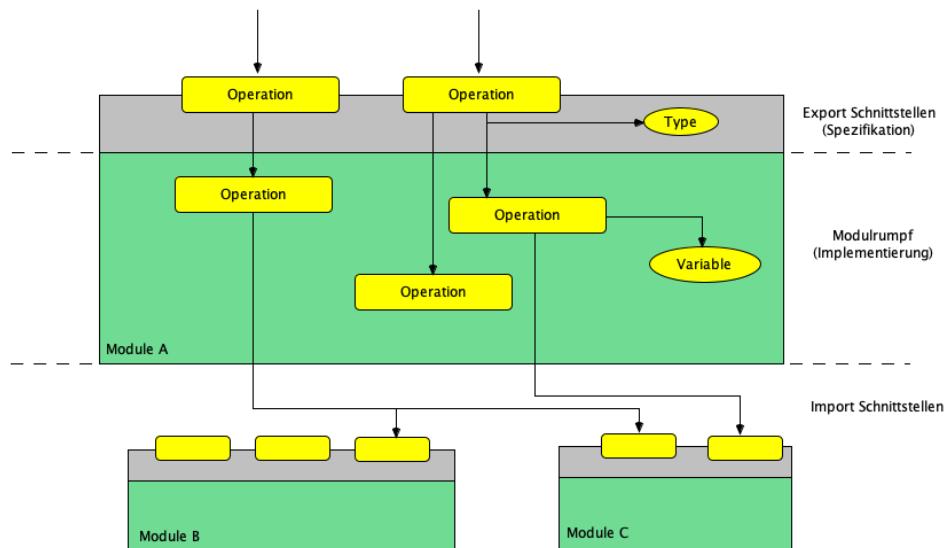


Abbildung 3.2: Schematischer Aufbau eines Moduls

In der Abbildung 3.2 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.

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 Programmsystem nachprüfbar.' [2]

Aus dieser Definition ergeben sich folgende Eigenschaften, 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 mit Hilfe 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

Mit Hilfe 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 Komponenten innerhalb eines Moduls zu messen, können die Abhängigkeiten in verschiedenen Kategorien eingeteilt werden: Logisch, Zeitlich, Prozedural, Sequentiell, Informal und Funktional.

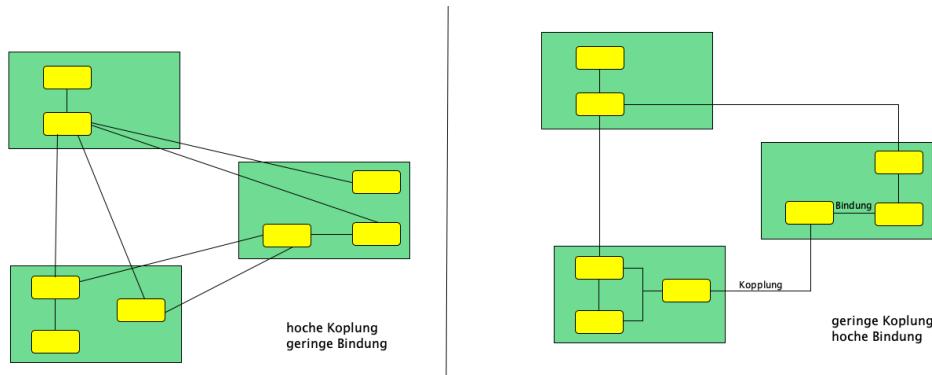


Abbildung 3.3: Modulbindung und Modulkopplung

Komplementär zu der *maximale 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 bringen Pakete wie *java.lang*, *java.io* und *java.net* mit sich. Andererseits gibt es die Benutzer konstruierten *Ap-*

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

Obwohl mit den vorher genannten *expliziten Module* Softwaresystemen realisieren lassen, fehlt die Offenheit bestimmter Module oder ihrer Pakete für die Umsetzung der Reflection Bibliotheken, die dynamischen Zugriff auf unsere Pakete während der Laufzeit benötigten. Wie im Kapitel 2.5 besprochen ist Reflection ein wichtiges Werkzeug in der Softwareentwicklung und wird in den neuen Java Modulsystem unterstützt. Um Reflection in einem Modul zu aktivieren, reicht es lediglich das ganze Modul als *open module* oder mit Hilfe des *opens package.name* Schlüssels ein spezielles Paket des Moduls in der *Modulbeschreibung* zu deklarieren. Damit hätte man ein für Reflection offenes Modul und könnte dessen öffentlich 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.

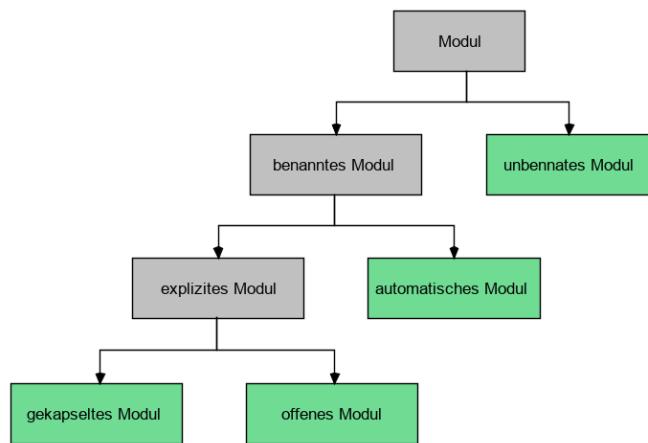


Abbildung 3.4: Modulararten

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 *un-*

benannte Modul beschreibt somit die 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 entstehen eine asymmetrische Kommunikation zwischen den Architekturen. Die *expliziten Module* arbeiten nur auf dem Modulpfad im neuen System und das *unbenannte Modul* 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.

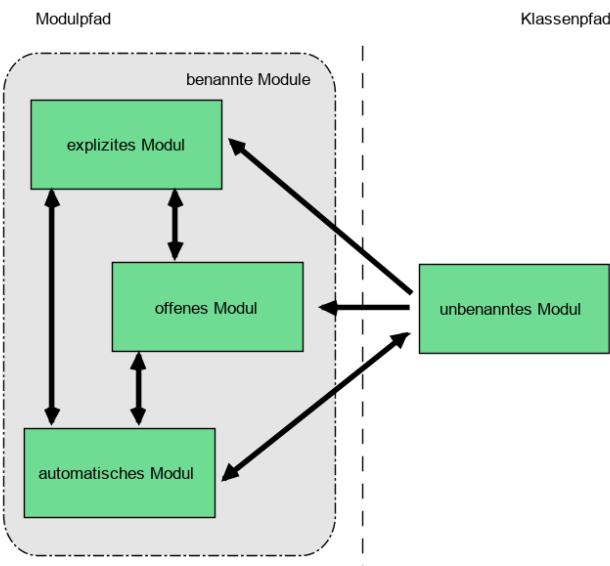


Abbildung 3.5: Modulzugriffsrechte

Das letzte Modul beschreibt ein Modul mit speziellen Verhalt, das sich zwischen den Architekturen stellt und eine Brücke zwischen den Modulpfad und Klassenpfad errichtet. Das *automatischen Modul* beschreiben einen Migrationsansatz der bestehenden Bibliotheken, die vom Klassenpfad auf den Modulpfad verschoben werden und keine *Modulbeschreibung* besitzen. Die-

se kriegen einen Modulnamen zugewiesen und können über diesen von den *expliziten Modulen* aufgerufen werden. Somit übernimmt Java die Kopplung der *automatischen Modulen* 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 den Modulpfad sowie den Klassenpfad. Dank dieser Fähigkeit können Bibliotheken migriert werden und beide Architekturen zu gleich unterstützen. Dieses Verhalten fördert die Entwickler ihren Code für den Modulpfad zu entwickeln, da die nötigen Legacy-Bibliothek der Applikation in beiden Architekturen zugleich verfügbar sind. Dennoch schafft das *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 bieten die automatische und das unbenannten 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 mit Hilfe der *module-info.java* Datei, die drei Kopplungstypen enthalten kann: die durch *requires*, *exports* und *opens* Deskriptoren beschrieben wird.

Die in der Abbildung 3.6 dargestellten und vorher diskutierten Kopplungsarten, können die Zugriffsberechtigungen ferner einschränken. Diese können ihr Schnittstelle exklusiven Module öffnen, die fortan als eine transitive Verbindung an gebundene Module 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.

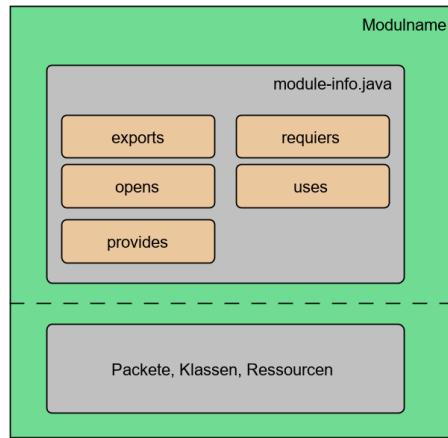


Abbildung 3.6: Die Schnittstellenbeschreibung *module-info.java*

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 -verwaltung geht über die Grundlagen hinaus und wird hier nicht weiter diskutiert, dessen ungeachtet ist es eine zusätzliche Möglichkeit die Modulkopplung zu minimieren.

Im Folgenden werden die 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

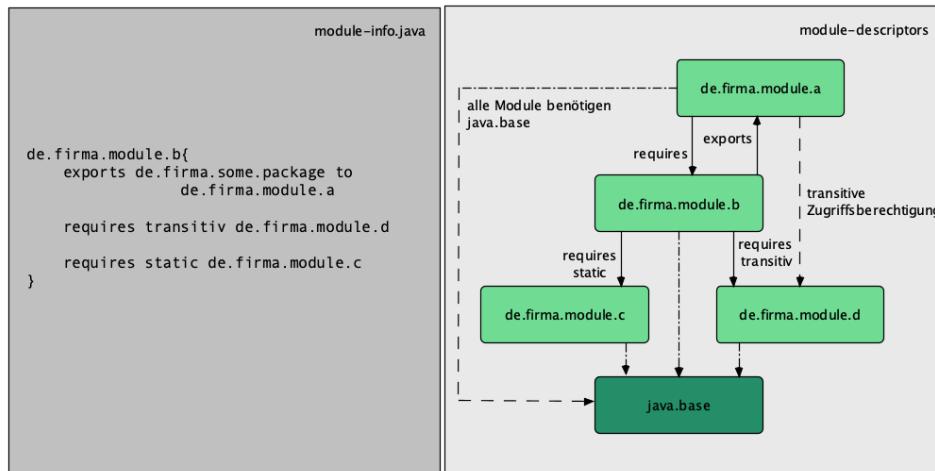


Abbildung 3.7: Abwandlung der Kopplungsarten

Wie in der grafischen Darstellung 3.7 abgebildet, handelt es sich bei den Kopplungstypen um Zugriffsrechte, die als ein offenes 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 2.3.3 wurde Namensräume vorgestellt, die Klassen von einemander trennen und diese als separate Software Komponenten behandeln, um die Sichtbarkeit der Codebasis gegenüber dem Restsystem abzugrenzen. Jedoch bring dieses Feature einen großen Aufwand mit sich, denn sofern die Applikation über eine große Anzahl an Bibliotheken nutzt und jedes davon auf einem eigenen Classloader betreiben möchte wächst der Wartungsaufwand mit der Anzahl der Bibliotheken.

Mit Hilfe der Module und dessen neuen Ansatz der internen Kapselung, soll dieses Problem adressiert werden, indem separate Zugriffsräume für jedes

Modul innerhalb eines Classloaders definiert werden, die sicherstellen, dass die interne Struktur eines Moduls während der Laufzeit nicht kompromittiert werden kann.

Um die Modulkapselung zu garantieren, wird das im Abschnitt 2.3.1 vorgestellte *Java Classloader 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 Classloader neue Fehlermeldungen wie die *IllegalAccessException* oder die *IllegalAccessError*. Somit bleibt die ehemalige Classloader Hierarchie erhalten, die an das Modulsystem von Java angepasst worden ist.

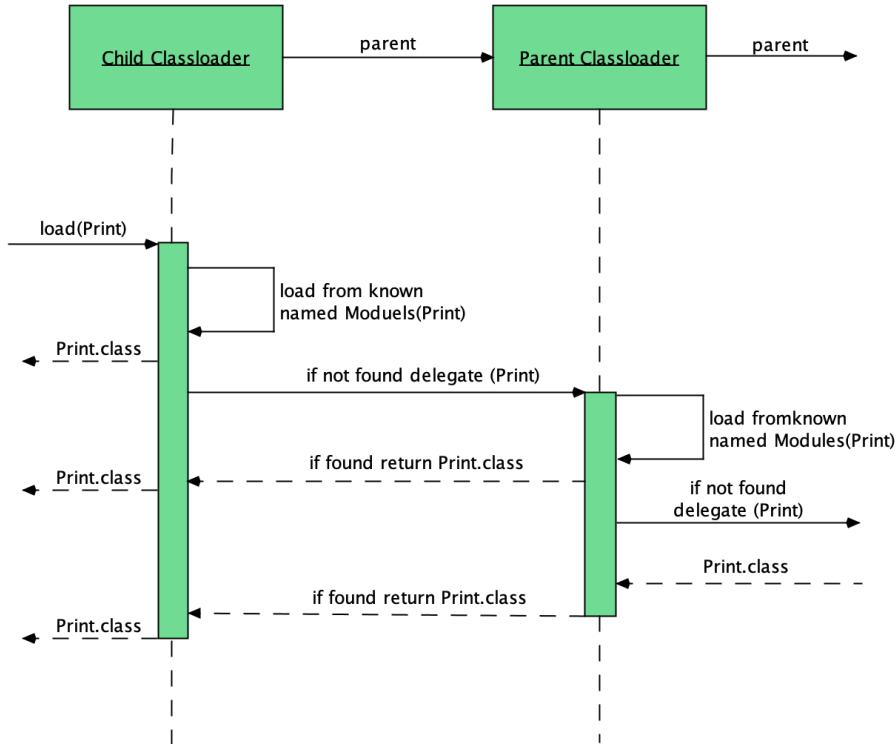


Abbildung 3.8: Modul Classloading

Da der JDK nicht mehr aus einer *rt.jar* besteht, sondern dessen Funktionalität auf Module aufgeteilt ist, wie in der Abbildung 3.9 abgebildet, beschäftigt sich das aktualisierte *Classloader System* mit dem laden bestimmter Module, die in Gültigkeitsbereiche eingeteilt sind. Diese müssen nicht mehr das über-

holte Delegierungsmodell 2.3.2 von Oben nach Unten durchsuchen, sondern suchen, die für den zuständigen Classloader, bekannten Module zuerst ab, bevor sie die Anfrage an den übergeordneten Classloader delegieren. Die Abbildung 3.8 visualisiert das Laden der Modul-Klassen, die in diesem Fall nicht mehr das unnötige Nachschlagen nach Klassen über die ganze Classloader-Hierarchie bearbeitet.

Indem das Delegierungsmodell abgelöst wurde, werden andere Sicherheitsmaßnahmen für das Schützen der Kern Bibliotheken benötigt. Dies übernimmt jetzt das Modulsystem von Java, indem ein globaler Verbot nach gleichnamiger Benennung von Paketstrukturen gilt. Somit verbietet Java zwei gleiche Bibliotheken auf dem selben Modulpfad und garantiert einen eindeutigen Abhängigkeitsgraphen.

Um die Klassen aus den Modulpfad zu laden, wird das *Classloader Systems* an den Modulpfad angepasst und lädt jetzt Module, die in Zugriffsgruppen eingeteilt sind. Der *Bootstrap Classloader* besitzt und geniest alle Sicherheitsprivilegien und steht demnach ganz oben in der Classloader Hierarchie. Dieser lädt die *Core Java-SE* und *JDK-Module*, wie *java.base* und *java.logging*. Der Extension Classloader wurde von dem Plattform Classloader 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 Classloader, der unsere Applikationsmodule auf dem Modulpfad verwaltet.

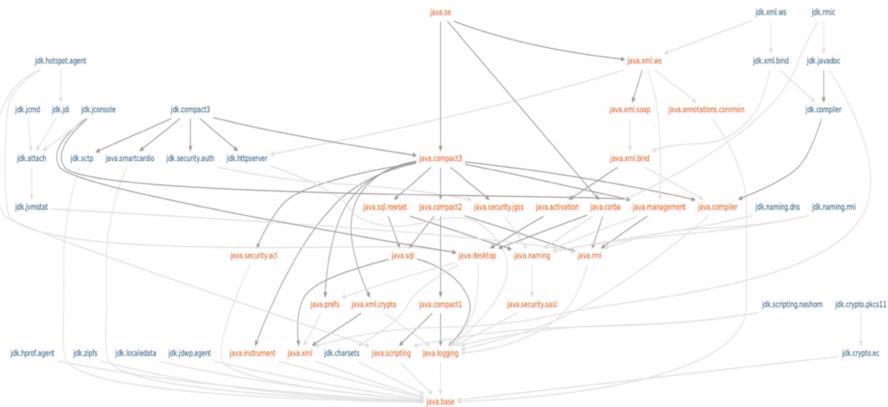


Abbildung 3.9: Modularisierte rt.jar Bibliotheken

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 dem selben Modulpfad.

Kapitel 4

Migration

Im vorherigen Kapitel wurden Module und ihre Eigenschaften, Konstruktionsregeln sowie Arten behandelt. Dieses Kapitel beschäftigt sich mit der Fragestellung, wie Altsysteme, die vor Java 9 entwickelt worden sind, auf dem Modulsystem betrieben werden können und was getan werden muss, um diese den modernen Anforderung anzupassen und vollständig zu Modularisieren.

Software die nicht beständig auf dem aktuellen Stand der Technik gehalten wird rutscht langsam in den Bereich der Altsysteme. Die Altsysteme werden oft als Legacy-Systeme bezeichnet, da diese lange Lebenszyklen besitzen und viele Code aus früheren Entwicklungszyklen mit sich tragen. Der Transfer dieser Systeme in eine neue Umgebung, ohne Änderungen der internen Struktur, bezeichnet man als Migration. Diese wird oft im Bereich der Softwaretechnik mit Software Reengineering und Software Neuimplementation verwechselt, dessen Zielsetzungen in der Optimierung der Codebasis liegen und nichts mit dem Ausführungskontext zu tun haben.

Die Migration von Anwendungen ist eine wiederkehrendes Ereignis im Lebenszyklus einer lang gepflegten Kernapplikation. 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.

Das Umfeld der Software Entwicklung ist eine dynamische Umgebung, denn auch mit einer gut durchdachte Architektur kann nicht garantiert werden, dass in der Zukunft heutige Paradigmen, Werkzeuge und Aufgabenbereiche den selben Kurs behalten. Deswegen existieren bereits sämtliche Migrationsstrategien, die als ein Leitpfaden den Entwicklung während der Migration führen.

Im folgenden Kapitel werden Ansätze vorgestellt, die Anwendungen aus dem monolithischen System in das modulare System überführen ohne Änderung an der interne Funktionalität durchzuführen.

4.1 Legacy-System

Der Begriff *Legacy-System* beschreibt ein altes System, das innerhalb einer Organisation länger als der implementierte 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. Die Entwicklung wurde von früheren Entwicklern und Teams durchgeführt. Damalige Konzeptentscheidungen ergeben ein Erbe, das für die zukünftige Erweiterung der Software eine große Rolle spielt. Legacy-Systeme wurden typischerweise gemäß der veralteten Praxis und Technologie entwickelt. Sie haben lange Lebenszyklen mit umfangreichen Veränderungen und Erweiterungen erfahren [3].

4.1.1 Eigenschaften

Bei Legacy-Systemen handelt es sich oft um sogenannte Kernsysteme, zur Unterstützung wesentlicher Geschäftsprozesse eines Unternehmens. Sie sind in der Regel geschäftskritisch und können nicht ohne größeren Aufwand und Risiko für das Unternehmen ausgetauscht werden. Aufgrund ihres langen Lebenszyklus, ihrer Komplexität und ständigen Überarbeitungen ist die Logik solcher Systeme oft unübersichtlich. Geschäftsprozesse und Geschäftsregeln sind im Code versteckt und müssten für z. B. eine Neuimplementation erst rekonstruiert werden [1].

Zu diesem Punkt gesellt sich die Eigenschaft, dass Legacy-Systeme oft sehr schlecht dokumentiert und strukturiert sind. Ihre Implementierung könnte

zudem früher geltenden Standards unterliegen und anderen Programmierparadigmen folgen, die nur schwer verständlich sind [4].

4.2 Migration

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 auf sich ändernde Anforderungen und Techniken des Anwendungsbereiches angepasst werden muss [3].

4.3 Migrationshürden

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

- Die Probleme bei dem Modulsystem beginnen mit den Zugriffsrechten auf die Core-JDK API's. Diese sind ab sofort in dem Module gekapselt und bieten keine Möglichkeit sie aufzurufen. 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 2.3.3, nämlich den Zugriff auf privat deklarierte Pakete aus fremden Modulen. Trotz dem gibt es Bibliotheken mit ähnlicher Paketstruktur die nicht böswillig sich Zugriff verschaffen wollen, sondern spiegeln eine Standardstruktur einer Bibliothek wieder, wie zum Beispiel *de.firma.input.reader* kann in mehreren Bibliotheken eines Unternehmens existieren und wird ab Java 9 nicht mehr zulässig sein. Somit müssen Module mit ähnlicher Struktur angepasst werden um den nächsten Modularisierungsschritt durchführen zu können.

- Der Classloader-Typ des Applikation-Classloaders wurden überarbeitet und infolgedessen auch das Arbeit mit den ehemaligen URLClassloader Methoden. Der bestehende Code, der den URLClassloader exzessiv nutzt und zum Beispiel Ressourcen aus verschiedenen Quellen lädt, muss auf den *SecureClassLoader* oder *ClassLoader* aufgewertet werden, um funktionstüchtig zu bleiben.
- Einer der Kritischen Veränderungen, die die Modultatform mit sich bringt, ist der Verbot von zyklischen Abhängigkeiten von Modulen untereinander. Diese dürfen sich nicht gegenseitig mit den Schlüssel *require* koppeln, da sonst eine Veränderung in einem Module zwangsläufig eine Änderung im anderen Module hervorrufen kann. Dieser Still 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 verbiete aus diesem Grund Zyklen in den Applikationsentwurf. Um bestehende Zyklen in einer Applikation zu lösen, muss die Funktionalität genau betrachtet und in kleine unabhängige Aufgaben aufgeteilt werden. Somit wäre der Zyklus aufgebrochen und die Aufgabestellung 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*, die zwei der gängigsten Arten der Softwaresystem Migration beschreiben.

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 Kommu-

nikation der Softwarebausteine muss über entsprechende Kommunikationskanäle errichtet werden und gestaltetet damit einen Brücke zwischen Alt- und Neuentwicklung, die zusammen einen gemeinsame Ressourcenbasis nutzen können.

Der *Butterfly-Ansatz* geht von einer separaten Entwicklung der Applikation in der neuen Umgebung aus. Dies hat zu 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 der 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 kompletten Applikation auf das Modulsystem weiterhin möglich, da diese keine spezielle Hilfsmittel von der Umgebung verlangt.

4.4.1 Plattform Migration

Die simpelste Migration ist eine reine Plattform Migration ohne Änderungen an der Software vorzunehmen, wie in der Abbildung 4.1 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 3.5 angesprochen, bietet dieses Modul den Betrieb der Legacy

Anwendung in der neuen Umgebung mit dem geringen Anteil der möglichen Vorteilen, wie zum Beispiel den Sicherheitsupdates. Zusätzlich behält die Applikation ihre monolithisch Architektur und vermisst alle Modulsystem Features die im Kapitel 3.1 und 3.3 angesprochen wurden.

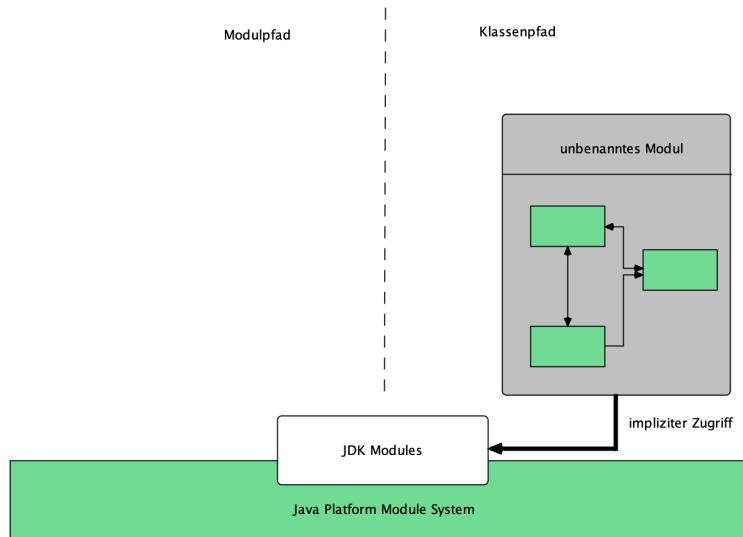


Abbildung 4.1: Plattform Migration

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 Stil migrieren, denn die Hürden aus den Kapitel 4.3, 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 3.4 besprochenen Kriterien erstellt werden. Mit dieser Methode werden die Drittanbieter Bibliotheken zuletzt betrachtet, da es bei diesen um Fremdcode handelt.

Der im ersten Schritt erstellter Abhängigkeitsgraph verweist ab einen gewissen Punkt die Bibliotheksabhängigkeit der Applikation, die wiederum von weiteren Bibliotheken abhängen. Somit können 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 3.5 behandelt, sowohl mit der Applikation als auch mit den Legacy Elementen ihrer eigener 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.

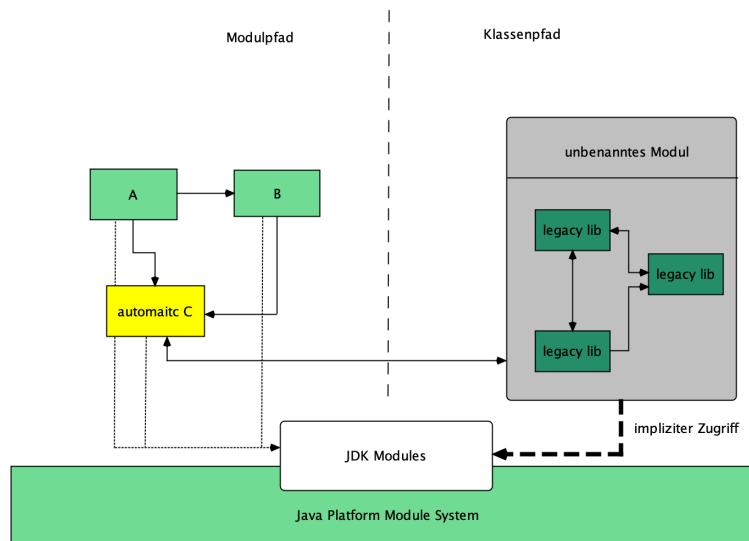
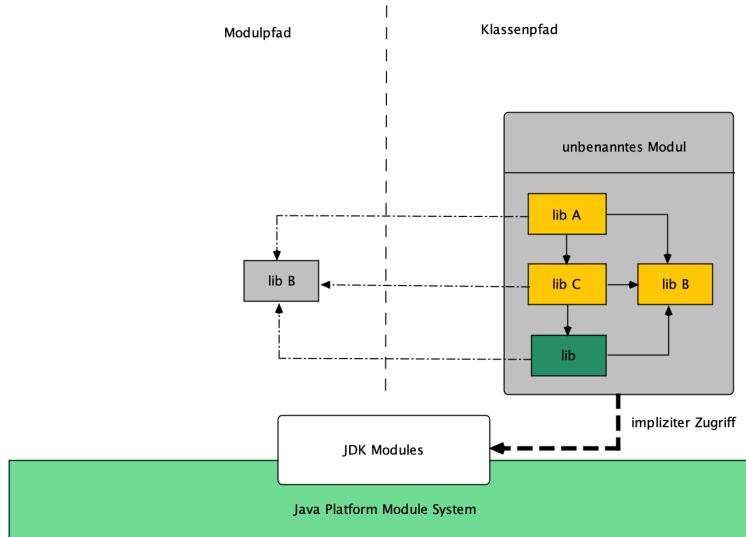


Abbildung 4.2: *Top Down* Migration

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 mit Hilfe des *JDepth* erstellt werden. Module


 Abbildung 4.3: *Bottom Up* Migration

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 3.4 besprochenen Kriterien erfüllt werden um eine robuste Umsetzung zu erreichen.

In der Abbildung 4.3 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 es eine Dritt anbieter 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 komplette 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 Applikations-

4.4. Migrationsarten

architektur 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

In diesem Kapitel geht es um die Anwendung des Modulsystems auf die Renew Applikation. Dabei soll die Umsetzung der Theoretischen Konzepte innerhalb der Applikation, die sich Langfristig bewährt hat, unverändert bleiben und zusätzlich mit den Modulsystem Eigenschaften ausgestattet werden.

Die initiale Entwicklung von Renew begann mit einer monolithischen Architektur. Diese erfüllte die nötigen Anforderungen, einengte sich jedoch nicht für Entwickler mit geringem Kenntnis der Gesamtarchitektur und der darunterliegenden theoretischen Konzepten. Daher wurde eine Plugin-Architektur aufgesetzt, die es ermöglichte Studenten Renew mit Logik und Plugins zu erweitern. Diese Trägt bereits den Gedanken der *Modularisierung* in sich, da die Gesamtarchitektur in Bestandteile zerlegt und mit einander entkoppelt verknüpft wurden. Mit der Einführung des Modulsystems wird der nächste Schritt in Richtung erweiterbare und zusammen setzbare Systeme vorgenommen.

Dieses Kapitel diskutiert die Motivation für die Migration, den Einfluss des Java Modulsystem auf Renew und die derzeitige Plugin-Architektur. Des weiteren wird ein Zustand ermittelt, der als Basis für die nachfolgenden Prototypen gelten wird.

5.1 Motivation

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

5.1.1 Verkürzter Entwicklungszyklus

Die Aufteilung einer monolithischer 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. Dies führt zu alten, unverständlichen Code aus der Java 1.4 (2002) Version, mit dem viele Konzepte und Architektur Entscheidungen getroffen wurden. Nach fast 18 Jahren Betrieb ändert die Codebasis sowie die Ideen und Konzepte für die Umsetzung ihrer Funktionalität. Besonders konfus und aufgeblättert können Funktionsumsetzungen erscheinen, die heute von Java 12 in ein paar Zeilen gelöst werden können. Der zügiger und rapider Wandel der Software Paradigmen und deren optimaler Einsatz in der Software Architektur ist ein Teil des Fortschritts und kann nicht ignoriert werden.

Daher 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 einen 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. So mit können Codeänderungen an mehreren Stellen im Plugin das Verhalten des Plugins beeinträchtigen und müssen auf das Gesamtverhalten getestet

werden. Mit der Einführung der Module, kann das Plugin in kleinere Einheiten zerlegt werden, die anschließend eine gekapselte Teifunktionalität des Plugins in sich tragen. Diese sind klein, leicht änderbar, ersetzbar und besitzen einen eignen Lebenszyklus. Somit verkürzt sich der Entwicklungsdauer einer Änderung und bietet eine Möglichkeit kooperativ und parallel an einem Plugin zu arbeiten.

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 keine komplette Gruppenaufmerksamkeit, um die passenden Codeblöcke für die Gesamtfunktionalität auszuwählen, da es so gut wie keine Überschneidung in der Aufgabenimplementation sich bilden kann. Somit profitiert Renew von den kurzen Entwicklungszyklen der Module und deren unproblematischen Verknüpfungseigenschaften.

5.1.2 Code-Bausteine

Eine der wichtigsten Fähigkeiten eines Entwicklers, ist die Beherrschung der Komplexität. Diese führt zu sauberen, lesbaren, wartbaren Code und erweitert den Lebenszyklus einer Software um ein Vielfaches. Um dieses 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-, Projekte- und Doktorarbeiten 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 an die gehalten werden muss, ein erstrebenswerte Charakteristik.

Die im Kapitel 3 vorgestellten Moduleigenschaften beschreiben die von dem Java Modulsystem eingesetzten Richtlinien für die saubere Softwareentwicklung und erzwingen ein Still der fein granulierte Code-Bestandteile, die kombiniert eine Softwaresystem darstellen. An erster Stelle verhindert dieses Vorgehen den sogenannten *Spaghetti Code*, der Funktionsübergreifende Anpassungen trifft und den Überblick über den Zusammenhang der Gesamtarchi-

tekur unscharf erscheinen lässt.

Module erschweren den *Spaghetti Code*, indem Mehraufwand für die Kommunikation zwischen den Modulen erbracht werden muss und machen das unsaubere Arbeiten unattraktiv. Somit dienen Module als Grenzen für den Entwicklungsrahmen eines Features und engen den Bearbeitungs- und Be trachtungsraum für den Entwickler ein. Daraus ergibt sich ein Softwarepaket, der unabhängig von den Senior-Entwicklern verstanden, genutzt und ange passt 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 immer fort wechselnden Akteur eine saubere Codebasis hinterlassen, die für den nächsten Absolventen sowie den wissenschaftlichen Mittearbeitern viel Zeit erspart.

Aus einer sauberen Umsetzung folgen saubere Code-Bausteine, die wieder verwendet werden können. Diese Eigenschaft der Module bringt ein wesent lich Vorteil beim Optimieren der Renew Applikation, indem kontextbezogen Module ausgetauscht werden können, um ein besseres, lokales Erlebnis 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 ver waltet und bei Bedarf ausgetauscht werden können.

5.1.3 Bereit für Innovation

Der zeitgemäße Zustand einer Applikation ist eine Zeichen hoher Qualität und reflektiert enorme Ansprüche an den Betrieb der Applikation. Diese kann geschäftskritische Qualitäten tragen, die den marktführenden Vorteile bringen und sich gegen die Konkurrenz durchsetzen. Um den Vorsprung zu sichern, ist eine *vorausschauende* Flexibilität gefragt. Mithilfe dessen die Applikation in der Lage ist, mit minimalen Aufwand, an die führenden Technologien anzuknüpfen.

Die Aktuell führenden Trends beschäftigen sich mit der verteilten und wiederverwendbaren 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 erreiche 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 interne 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 unseren 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 den Zusammenspiele von *Applikationsschwärme* kümmert.

Mit Hilfe 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 integriert wird.

5.2 Ausgangssituation

Renew ist in mehr als 60 Plugins aufgeteilt, die für sich allein stehende Projekte repräsentieren. Jedes Projekt besitzt eine *buidl.xml* und wird mit dem übergeordneten Stamm *build.xml* Script zusammengeführt. Die XML-Scripte werden von dem *Apache 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 nötige Plugin-Abhängigkeiten und wird von den 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.

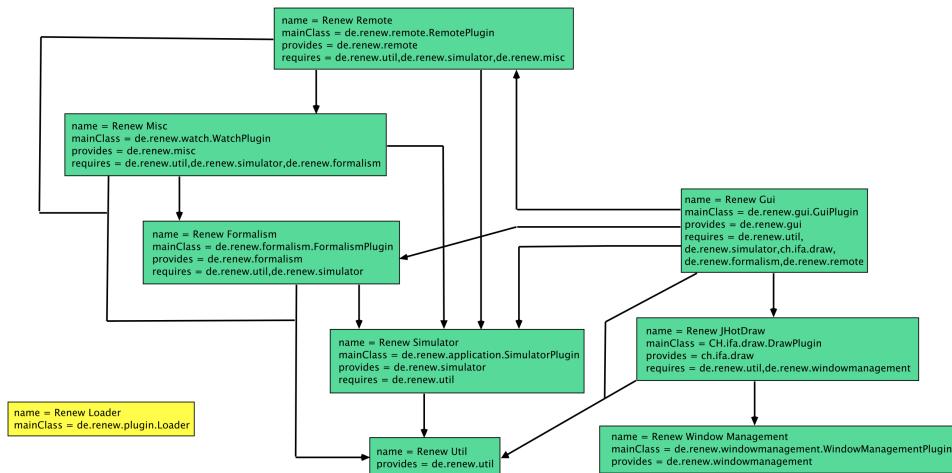


Abbildung 5.1: Gui Plugin-Abhängigkeiten

Für die Evaluation der minimale Konfiguration starten wir aus dem *Gui*-Plugin und arbeiten uns abwärts der Plugin-Hierarchie der *plugin.cfg* hinab,

bis der komplette Graph aufgebaut ist.

Die in der Abbilde 5.1 repräsentierten Zusammenhängen reflektieren die von den Entwickler abgestimmten Laufzeitabhängigkeiten, die einen groben Überblick über die nötigen Plugins verschaffen. Diese können zur Laufzeit alle benötigen Daten und Klassen enthalten, jedoch tragen sie keine Aussage über Abhängigkeiten während der Kompilation. Demgemäß kann zusätzlicher Code sowie Plugins benötigt werden.

5.3 Auswirkung

Die Folgeerscheinung der Modularisierung unterbindet zahlreiche Entwicklungsschwächen, wie *Zyklen*, *Split Packages* und antiquierte API's. Diese sollen mit der Migration aufgelöst, reorganisiert und nachgerüstet werden, um einen kompierfähigen Zustand erreichen zu können.

5.3.1 Zyklenfrei

Mit den Zyklen-freien Plugins werden Abhängigkeiten aufgelöst, die sich an falschen stellen befinden und mehr als eine Aufgabe Plugin übergreifend lösen möchten. Diese haken sich in den Betrieb der unmittelbar angrenzenden Komponenten ein und machen sich unverzichtbar für die Ausführung der Software. Die minimale Renew Version hat keine Laufzeitzyklen und erfüllt somit das Kriterium der Zyklenfreiheit, nichtsdestotrotz sind diese in der erweiterten Version nicht ausgeschlossen.

5.3.2 Split Packages

Der Mangel der namensgleichen Pakete wird in einer Plugin-Architektur mit großer Wahrscheinlichkeit einräten, da Plugins aus dem gleichen Kontext, wie *Navigator* oder *NavigatorGit*, den selben *de.renew.navigator.** Namensraum beanspruchen. Dieses führt zum Fehler beim auflösen der benötigten Klasse und verhindert das Hochfahren der Applikation.

5.3.3 Schnittstellen

Die *plugin.cfg* beschreibt die Abhängigkeit der Module unter einander, jedoch ist diese Informantin unvollständig, denn dieser fehlt die Information

über Schnittstellen des Plugins und dafür ausgelegten Pakete. Infolgedessen hat der Entwickler keine andere Möglichkeit als auf gewünschte Plugin-Funktionen direkt zu zugreifen ohne einen festen Vertrag über die Kommunikation einzugehen. Somit kann jede Änderung und Aktualisierung innerhalb eines Plugins zum unerwarteten Verhalten seiner Nutzer führen.

Im Gegensatz zu den *plugin.cfg* führt die *module-info.java* eine Liste an benötigten Modulen sowie exportierten Paketen, die den Zugriff von Außen einschränken. Dies erlaubt nicht nur das Verständnis der benötigten Module, sondern unterstützt den Entwickler durch klar definierte Schnittstellen, die mit dem Schlüssel *exports* deklariert sind.

5.3.4 Homogene Umgebung

Einer der am weit verbreitetsten Probleme der Softwareentwicklung, ist der Unterschied der Laufzeit- sowie Entwicklungsumgebung. Die Entwickler arbeiten auf einer bestens eingerichteten Maschine mit allen nötigen Bibliotheken und Abhängigkeit, die in der Zielumgebung zu meist nicht existieren. Um die Diskrepanz des Ausführungskontext entgegen zu wirken, wird eine Richtlinie benötigt, die den Kontext einer Applikation beschreibt. Aus diesem Grund deklariert die *module-info.java* eine globale Plattformunabhängige Anforderung an benötigten Module und Bibliotheken, die zu der Kompilations- und Ausführungszeit präsent sein müssen.

Die Renew *plugin.cfg* erfüllt die Funktion einer Richtlinie der benötigten zu grunde liegender Plugins, jedoch fehlen die Deklaration der unterstützenden Bibliotheken, die für die Ausführung von Renew benötigt werden. Daher ist die explizite Deklaration der notwendigen Drittanbieter-Bibliotheken wie *log4j* eine willkommene Erweiterung für Renew.

5.3.5 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 von einander und erlaubt das einfache hinzufügen sowie entfernen von Plugins ohne die bestehende Architektur zu verändern. Die notwendigen Werkzeuge, die im Grundlagenkaltteil besprochen wurden ermöglichen die Umsetzung der dynamische Plugin

Kopplung.

Obwohl die Plugins Unabhängig von einander entwickelt werden können, musste die interne Funktion an die umgebenen Plugins durch direkte Zugriffe gebunden werden. Daraus folgt eine wilde Kopplung jedes einzelnen Plugins mit der umgebender 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 ergibt 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, mit Hilfe des *provide with* Schlüssels eine Implementation für das *Formalism*-Modul anbieten. Diese werden anschließend über die *Java Servide Loader Registry* von dem *Gui*-Plugin ausgelesen und verwaltet.

Die Instantiierung der Plugins über das *IPPlugin* 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 .

5.3.6 Plugin Manager

Der *Plugin Manager* ist verantwortlich für das Erfassen und Einlesen der Renew Codebasis in der Arbeitsspeicher. Die Reihenfolge für die Plugin-Architektur ist strikt, da bestimmte Plugins gewisse Drittanbieter sowie Plugin-Bibliotheken brauchen. Wenn die Reihenfolge nicht stimmt werden benötigte Klassen nicht gefunden und die Applikation bricht mit einem Fehler ab. Daher ist das Laden der Drittanbieter-Bibliotheken und das anschließende Laden der Plugins in der richtigen Reihenfolge eine kritische Aufgabe.

Das Modulsystem von Java bringt einen neuen Abschnitt in den Lebenszyklus einer Java Applikation, der diese wichtige Aufgabe des *Plugin Managers* übernimmt. Jener erstellt einen gerichteten Graphen aus angeforderten Modulen sowie Bibliotheken und prüfte dessen Existenz auf dem Modulpfad. Wenn diese nicht existieren wird ein *java.lang.module.FindException* Fehler geworfen, anderenfalls sind alle Voraussetzungen für das Starten der Applikation erfüllt und die Applikation beginnt ihre Ausführung.

Damit übernimmt das Modulsystem von Java die Erstellung der richtigen einlese Reihenfolge der Plugins als einen gerichteten azyklischer Graphen und übernimmt eine weitere interne Aufgabe.

Kapitel 6

Prototypen

In diesem Kapitel entstehen Prototypen, die Renew schrittweise modularisieren bis die Applikation den größten Teil ihre Funktionalität auf dem Modulpfad betreiben kann.

Für die Umsetzung 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 liegender Logik **ausstattet** ist.

6.1.2 Projektstruktur

Für die Umsetzung des modularen Renew's 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 binden** der Codebausteine geschieht versteckt und für den Entwickler schwer zugänglich. Es liegt ein weiter und verschachtelter **Weg der Eclipse Konfiguration-UI**, die sich mit der Zeit verändern kann. Dieser Umstand wurde von mir im letzten Projekt beobachtet und kostete Zeit für alle Projektteilnehmer, da die Universitätsrechner strikten Rechten unterliegen, die keine **eigen** 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 die Umstrukturierung von Renew an das Modulsystem durchgeführt werden muss, muss das für die Kompilation und Verpacken der Codebasis verantwortliche Werkzeug die Veränderung miterleben.

Renew benutzt zur Zeit das *Apache Ant* Werkzeug, dass alle Plugins kompiliert und in einer ausführbare Form bringt. **Dieses** ist in Jahre gekommen

und enthält wesentlich geringeren Funktionsumfang gegenüber der Aktuellen Konkurrenz, wie Maven und Gradle. Diese bieten eine Abhängigkeitsverwaltung, konfigurierbare Plugins und sogar eine Programmiersprache. Im Gesetz zu der aufgeblasenen XML-Konfiguration von Ant, beherrschen die modernen *build* Werkzeuge die Komplexität durch den *Convention over Configuration* Ansatz und flexiblen Ausdrucksweisen.

Die minimalen Version von Renew soll sich an einem modernen *build* Werkzeug bedienen und eine 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 mit einem aufstrebenden Werkzeug.

Dafür muss das bestehende Ant *build* System analysiert und mit dem Gradle Werkzeug aufgebaut 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 ein bekannte Ordnerstruktur mit Drittanbieter-Bibliotheken, Plugins und Konfigurationsdateien, die über den *Plugin Manager* 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.

Für den ersten Prototypen wird zuerst eine Projektstruktur erstellt die für jedes Plugin Projekt die Möglichkeit bietet aus mehreren Modulen zu bestehen. Dafür wird eine Struktur 6.1 erstellt, die im Java Verzeichnis alle Module bündelt, die über den Modulnamen disjunkt von einander 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.

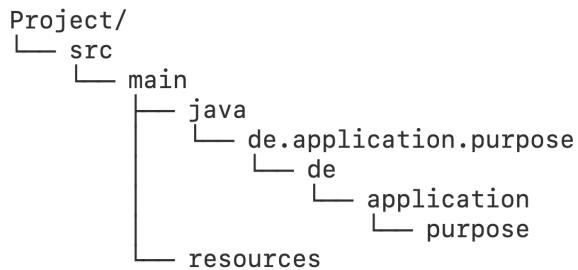


Abbildung 6.1: Projektstruktur

Nachdem die Projektstruktur unseren Wünschen entspricht, muss diese in der Gradle Konfigurationsdateien verankert werden. Hierfür halten wir für jedes Projekt die Projektstruktur und dessen Abhängigkeiten in der *build.gradle* Konfigurationsdateien fest, indem wir Java- sowie Ressourcen-*sourceSets* definieren und Projekt sowie Drittanbieter-Bibliotheken Abhängigkeiten für den Kompilation-Pfad bestimmen.

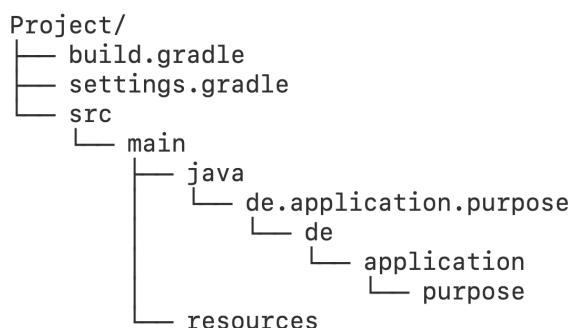


Abbildung 6.2: Gradle Konfiguration

Die oben genannten Schritte müssen für jedes Projekt, das für die minimal Version von Renew auserwählt wurde durchgeführt und im Anschluss über die entsprechende Entwicklungsumgebung validiert werden. Wenn diese alle

Klassen und die benötigten Abhängigkeiten finden und kompilieren kann, haben wir alle Projekte richtig strukturiert, definiert und mit einander sauber verbunden. In diesem Zustand ist die komplette Struktur des Projekts innerhalb 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 in dem Kapitel Migration 4.4.3 vorgestellten *bottom up* Ansatz verwenden.

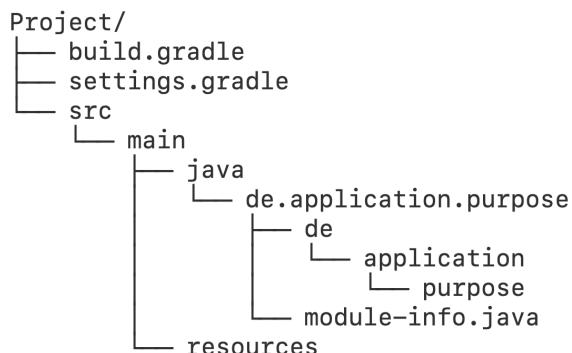


Abbildung 6.3: Modulumwandlung

Zuerst werden die Drittanbieter-Bibliotheken, wie *log4j*, auf den Modulpfad als automatische Module eingebunden und werden somit 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, wie zum Beispiel das *Util* Plugin. Damit diese auf dem Modulpfad ausführbar sind, werden sie durch eine *moduel-info.java* Konfigurationsdatei erweitert. Diese muss sich im Stammverzeichnis des Moduls befinden wie in der Abbildung 6.3 dargestellt und deklariert die benötigten automatischen Module. 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

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.

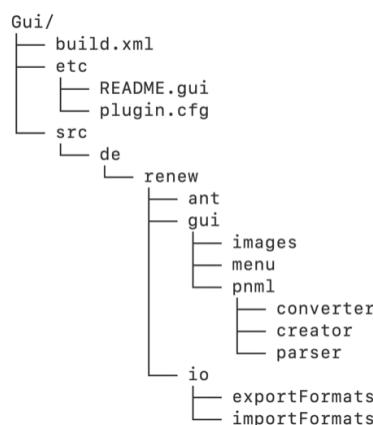


Abbildung 6.4: Gui Projekt

Das Gui Plugin **eröffnet** die geläufige mangelnde Organisation der Ressourcen. Zum Teil befinden sich diese in den *etc* Verzeichnis und zum Teil sind diese in den Java *source set* im *images* Verzeichnis integriert, wie in der Abbildung 6.4 dargestellt.

Um diese zu beheben, wird das **Bild** 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.

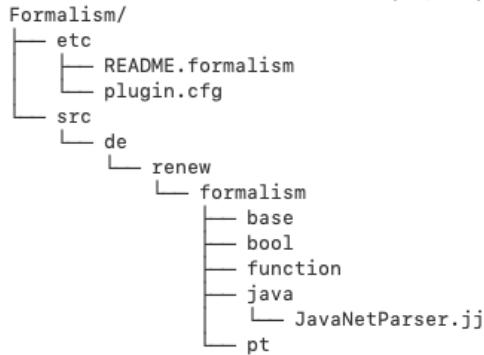


Abbildung 6.5: Formalism Projekt

Andere Plugins wie Formalism, CH oder Misc besitzen *JavaCC* Dateien, wie im Beispiel 6.5 dargestellt **enden** diese auf *jj*. Sie erstellen Java Netz Grammatiken und wandeln die Java-Basis für die Ausführung ab. Diese liegen lose zwischen den Java Klassen und werden von den Java Compiler nicht interpretiert. Daher macht es Sinn diese in ein eigens *Source Set* auszulagern und für den *JavaCC* Compiler für die Übersetzung zu gruppieren.

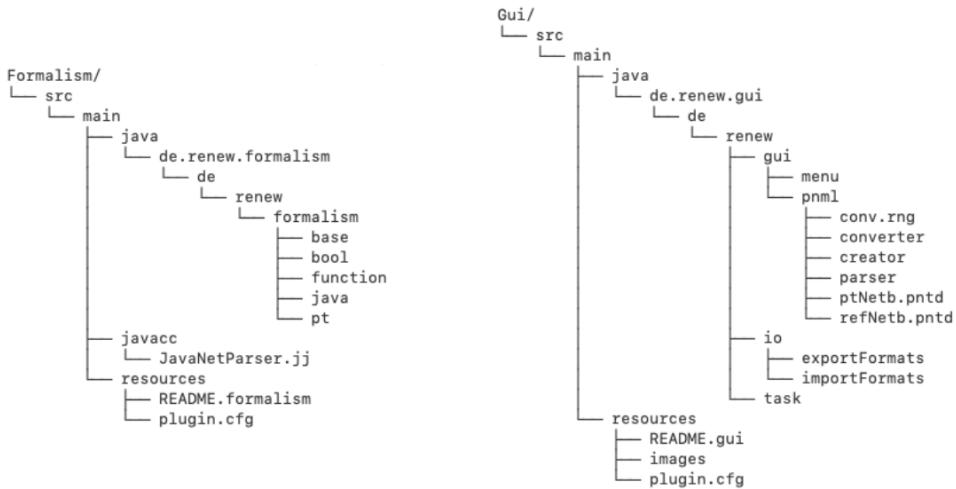


Abbildung 6.6: Resultierende Projektstrukturen

Das Resultat 6.6 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* zum Gunsten 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 fallen.

6.4.2 Gradle

Um die Zyklen zu erkennen, wird ein Gradle *build* Skript erstellt, der die Java *Source Sets* für den Kompilation Schritt definiert und die benötigten Projekte sowie Drittanbieter-Bibliotheken auf den Projektklassenpfad einbindet. Dafür wird ein übergeordneter Gradle Projekt deklariert, der alle Subprojekte aus allen Plugin Verzeichnissen erstellt.

```
rootProject.name = 'renew'

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

Abbildung 6.7: Subprojekte

Die *settings.gradle* Datei in der Abbildung 6.7, 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/$moduleName"]
            java.outputDir = file("$buildDir/modules/$moduleName")
        }
    }
}
  
```

Abbildung 6.8: 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 wo Verzeichnisse mit den Java Code und den dazugehörigen Ressourcen sich befinden sollen.

```

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

Abbildung 6.9: 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 von uns im GitLab Repository bereitgestellt werden. Die deklarierten Drittanbieter-Bibliotheken in der Konfiguration erleichtern zusätzlich die Aufgabe der manuellen Erstellung der Klassenpfade und die Einbindung der Bibliotheken in der Entwicklungsumgebung, da die Konfiguration von der Entwicklungsumgebung aufgegriffen und auf das Projekt angewandt wird.

```

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

Abbildung 6.10: Klassenpfade

Um die Klassenpfade von einander zu trennen werden zusätzliche Konfigurationen mit dem Namen *plugin* und *automaitc* eingeführt, die Plugin Code und Drittanbieter-Bibliotheken von einander trennen und für das Kompilieren zusammenführen. Somit könne diese getrennt von einander Verwaltet, Modifiziert und bei Bedarf für bestimmte Aufgaben angepasst werden.

```

jar {
    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 6.11: 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.

```

ext.moduleName = 'de.renew.windowmanagement'

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

Abbildung 6.12: Individuelle Konfiguration

Die Plugins benötigen einen internen Namen für die Verwaltung und die zusätzlichen Drittanbieter-Bibliotheken sowie Plugin Abhängigkeiten. Dafür wird in der Plugin *buidl.gradle* Konfigurationsdatei der Name unter den *extension properties* deklariert und die bereits geerbten Abhängigkeiten erweitert. In der Abbildung 6.12 wird das WindowManager Plugin durch eine lokale, modifizierte Drittanbieter-Bibliotheken und durch das Plugin Projekt erweitert, um alle Benötigen Abhängigkeiten abzudecken.

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 definieren. Aus diesen wird ein Abhängigkeitsgraph erstellt, der Zyklen und versteckte Abhängigkeiten offenlegt.

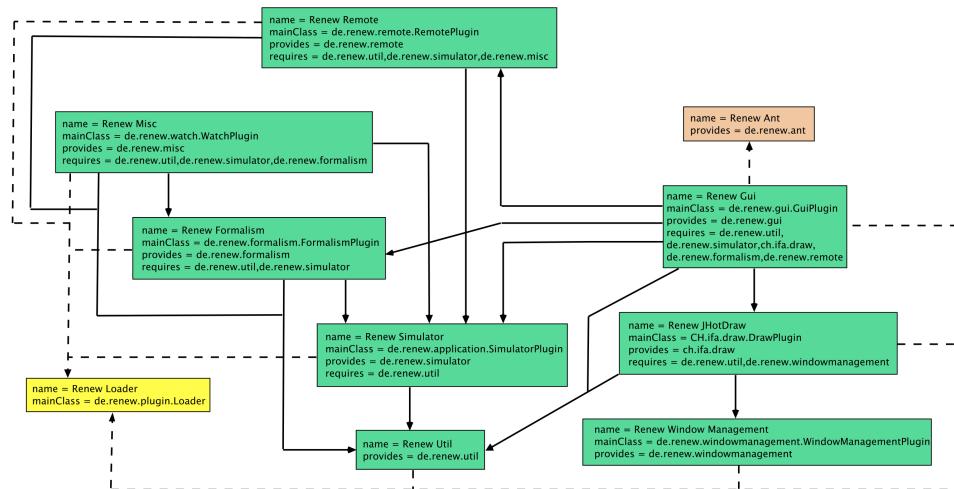


Abbildung 6.13: Kompilation Abhängigkeiten

Der neu entstandenen Graph wurde im Vergleich zu dem Graphen aus dem Ausgangssituation Abschnitt 5.1 durch ein Plugin erweitert und bindet alle

Plugins an das *Loader* Projekt. Des weiteren sind auf der Abbildung 6.13 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 Klassenpfade 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 migrierten Plugins, wie bereits in den Abschnitt 3.5 beschrieben wurde.

Da ein Modul seine eigne 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 den Klassenpfad auf dem Modulpfaden als automatische Module aufgesetzt.

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

Abbildung 6.14: Module Infos

In diesem Zustand wird Renew kompiliert und ausgeführt. Das Ergebnis ist eine lauffähige Applikation, die identisch zu der initialen minimal Version von Renew funktioniert.

Im nächsten Schritt werden Plugins migriert, die nur auf die neu entstandenen Module aufsetzen, wie zum Beispiel das *Simulator* und das *JHotDraw*

Modul. 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 *requiers* Schlüssel aus den Loader, Util und Windowmanagement Plugins. Um diese Anforderung zu entsprächen, 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 {
    // Renew
    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 6.15: Module Infos

In der Abbildung 6.15 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 alle Plugins auf dem Modulpfaden befinden, durchgeführt werden. Nähmlich 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.

Auf der Nächsten Seite in der Abbildung 6.16 wird die komplette Migration in vier Schritten dargestellt.

6.4. Umsetzung

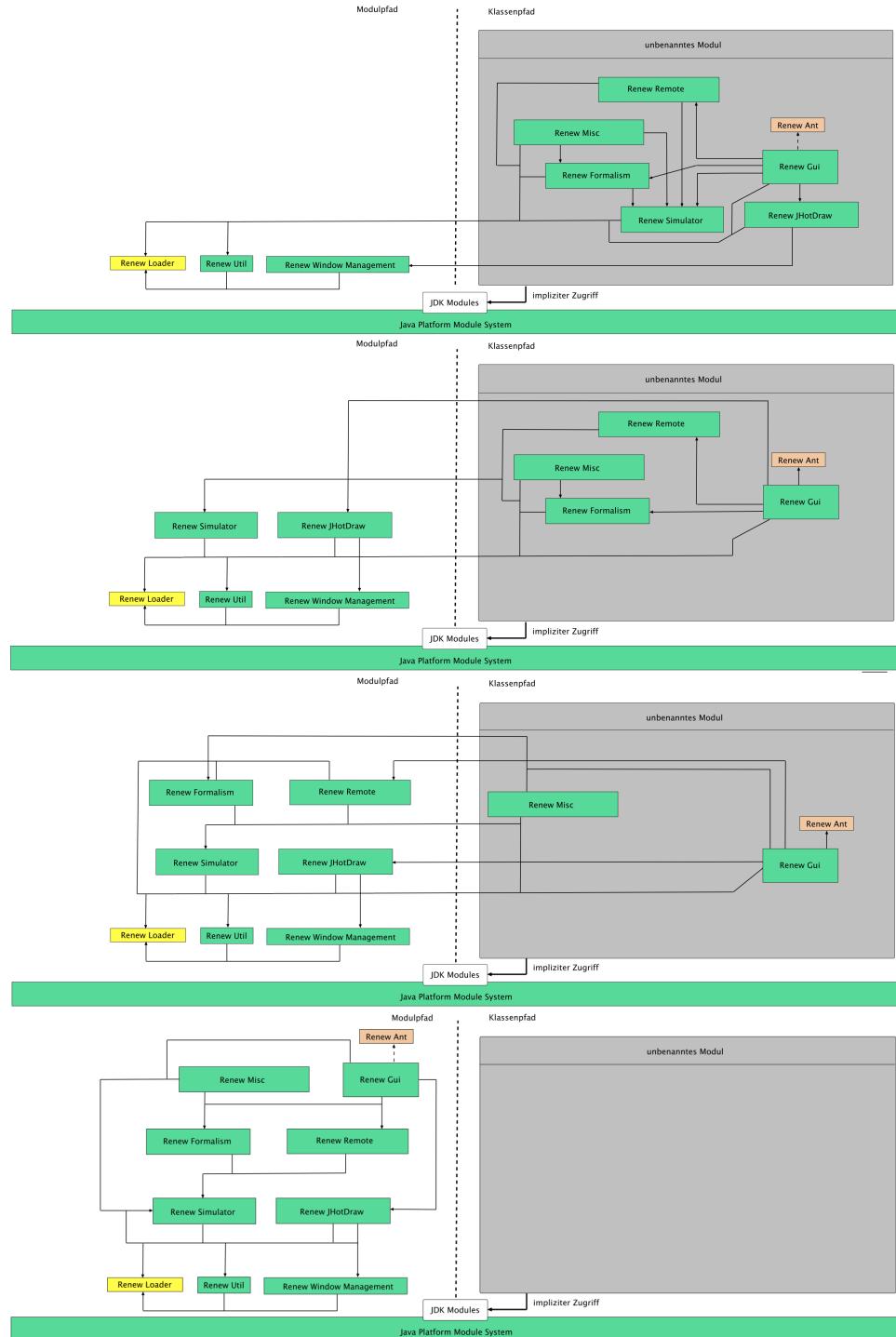


Abbildung 6.16: Migration

6.5 Evaluation

6.5.1 Struktur

Die Anforderungen 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 zusätzliche 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 Ressourcen, 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 zu 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 zwingen erforderlich für die Kompilation. Dies bezüglich wurde der *Gradle Task Graph* modifiziert, um die die entsprechende Reihenfolge zu erfüllen. Daraus ergibt sich 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 Entwickler in die Struktur der genutzten Bibliothek. Somit wird viel Zeit gespart, da man sich mit dem Kontext und der 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 benutzerdefinierte Logik befreit werden.

Ein zusätzlicher Vorteil der neuen Projektverwaltung liegt an der neuen Umsetzung mit einer Programmiersprache, die es erlaubt mit Felder, Variablen, Schleifen und Objekten zu arbeiten. Dies hat zu 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.

Die Verständlichkeit des Gradle Werkzeug gegenüber dem Ant Werkzeug spiegelt sich in der Code Menge, die geschrieben werden muss **wider**. 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 6.17: 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 Aufwertung

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

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

Abbildung 6.18: Statische Konfiguration

Des weiteren wird klar, dass die Renew Applikation in einer hierarchischen Plugin Architektur aufgebaut ist. Dass 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 genutzten Schlüsselmodule zu behalten, kann mithilfe der **transitiv** Deklaration eine angeforderte Bibliothek für alle Nutzer-Plugins offen gelegt werden. Demzufolge kann ein Plugin nicht nur ein anderes 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 6.19: Transitive Konfiguration

Darüber hinaus unterstützt Gradle die Idee der Modularisierung von Java und bietet eine unterstützende API zum entwerfen der transitiven Abhängigkeiten, indem zwei weitere Konfigurationspfade angeboten werden, 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 Bibliothek 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 6.20: Transitive Gradle Konfiguration

6.5.4 Endergebnis

Wenn man die Verfeinerung durchführt, kriegt man 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.

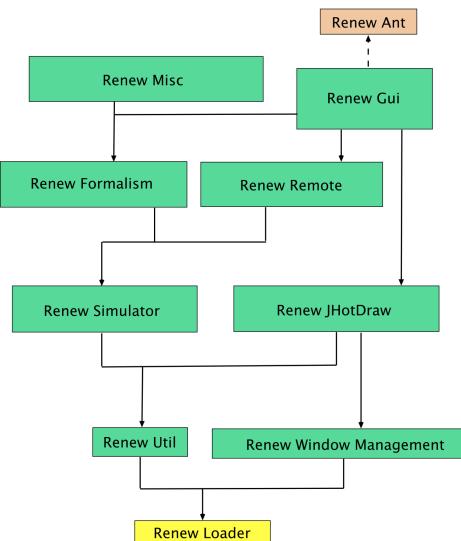


Abbildung 6.21: Minimale modulare Renew Version

Literaturverzeichnis

- [1] Andreas Martens. Ablösung von legacy-systemen in zeiten des digitalen wandels. *Wirtschaftsinformatik & Management*, 8(6):32–41, 2016.
- [2] P. Rechenberg. *Informatik-Handbuch*. Hanser, 2006.
- [3] Harry M Sneed, Heidi Heilmann, and Ellen Wolf. *Softwaremigration in der Praxis: Übertragung alter Softwaresysteme in eine moderne Umgebung*. dpunkt. verlag, 2016.
- [4] Peter Stahlknecht and Ulrich Hasenkamp. *Einführung in die Wirtschaftsinformatik*. Springer-Verlag, 2002.