technische universität
dortmund

Fakultät für
Informatik

Masterarbeit

Extraktion und Visualisierung von Beziehungen und Abhängigkeiten zwischen Komponenten großer Softwareprojekte

Tobias Marquardt

25. April 2016

Institution

Technische Universität Dortmund
Fakultät für Informatik
Lehrstuhl 5 für Programmiersysteme

Gutachter

Prof. Dr. Bernhard Steffen
Dipl.-Inf. Stefan Naujokat

In Kooperation mit

Deutsches Zentrum für Luft- und Raumfahrt
Simulations- und Softwaretechnik

Inhaltsverzeichnis

1 Einleitung

Die Architektur von Software ist für Menschen nur mit Hilfe des Quellcodes schwer zu erfassen. Deshalb ist das Ziel dieser Masterarbeit, mittels visueller Darstellungen ein besseres Verständnis von Softwarearchitekturen zu ermöglichen. Der Fokus liegt dabei auf OSGi-basierten Java-Anwendungen, welche Merkmale besitzen, die sie für dieses Anliegen besonders interessant machen. Betrachtet werden vor allem Abhängigkeiten zwischen Komponenten der Software. Doch auch andere Aspekte und Metriken werden miteinbezogen. Es wird eine Implementierung vorgestellt, die benötigte Daten automatisiert sammelt und mit geeigneten Visualisierungsverfahren interaktiv darstellt. Am Beispiel der Software „Remote Component Environment“ vom Deutschen Zentrum für Luft- und Raumfahrt wird das Ergebnis evaluiert.

1.1 Motivation

Schon vor Jahrzenten, sagten Experten voraus, dass sich die Tätigkeit des Programmierers in Zukunft stark wandeln wird [80]. Weg vom manuellen Schreiben von imperativem, maschinenorientiertem Code, hin zur deklarativen Spezfikation auf hoher Abstraktionsebene, domänenspezifischer Modellierung [69], grafischer Programmierung [51] und letztendlich automatischer Generierung von Programmen. Trotzdem ist das Schreiben von Quellcode in „General Purpose Languages“, wie Java und C++, immer noch die gängigste Methode Software zu entwickeln. Doch bei der Betrachtung und Analyse stößt solcher Quellcode als Darstellungsart schnell an seine Grenzen. Besonders in großen Projekten mit hunderttausenden Code-Zeilen ist eine Betrachtung auf einer höheren Abstraktionsebene nötig, um sich einen Überblick zu verschaffen und Zusammenhänge zu verstehen [45]. Softwarearchitektur, die an sich unsichtbar und ein immaterielles Konzept ist, kann durch grafische Darstellungen und visuelle Metaphern fassbarer und verständlicher gemacht werden [62].

„Softwarevisualisierung“ ist zwar ein mittlerweile großes Forschungsgebiet [23], an praxistauglichen Werkzeugen mangelt es aber dennoch. Das trifft besonders auf OSGi-basierte Anwendungen zu, obwohl diese oft einen großen Umfang haben, bei dem Visualisierung besonders hilfreich wäre. Im Vergleich zu „normaler“ Java-Software, beinhalten OSGi-Anwendungen zusätzliche Aspekte, die bei der Betrachtung der Architektur eine bedeutende Rolle spielen. Dazu zählen vor allem die Konzepte von Bundles und Services, die eine Modularisierung des Codes unterstützen. Bisher gibt es allerdings wenig bis gar keine Analyse- und Visualisierungssoftware, die diese Aspekte berücksichtigt.

Nicht nur aus diesem Grund sind klassische Darstellungen wie Klassendiagramme zur Betrachtung begrenzt geeignet. Große Softwareprojekte bestehen aus mehr als nur dem reinen Quellcode. Möchte man eine Software in ihrer Gesamtheit er-

fassen, müssen auch andere Informationen einbezogen werden. In OSGi-Projekten gibt es neben dem Java-Code zum Beispiel andere Textdateien, die deklarativ in einem XML-Format wichtige Daten zur Architektur der Software enthalten. Doch auch externe Tools, die im Softwareentwicklungsprozess eingesetzt werden, können relevante Metadaten bereitstellen. Versionskontrollsysteme und Bug-Tracker ermöglichen zum Beispiel eine erweiterte Sicht auf ein Projekt, die der Quellcode alleine nicht liefern kann. Die Kombination dieser verschiedenen Datenquellen bietet ein großes Potential hinsichtlich der Softwarevisualisierung.

1.2 Ziele und Aufgaben

Ziel dieser Arbeit ist es, Visualisierungen für OSGi-basierte Java-Software zu entwickeln, die einem Entwickler beim Verständnis der Architektur helfen und einen Mehrwert für den Softwareentwicklungsprozess liefert.

Dafür soll zunächst analysiert werden, welche Aspekte der Software bei der Visualisierung von Interesse sind. Es können dynamische oder statische Eigenschaften betrachtet werden. Während erstere nur zur Laufzeit des Programms ermittelt werden können, reicht für letztere die Analyse des Quellcodes oder anderer statischer Artefakte, die zur Compile-Zeit vorliegen, aus. Desweiteren können evolutionäre Aspekte miteinbezogen werden, also Informationen darüber, wie sich die Software über die Entwicklungszeit verändert.

Bei der Auswahl der Aspekte, welche Teil der Konzeptionsphase ist, müssen die Zielsetzung und die Zielgruppe(n) der späteren Visualisierung berückstichtigt werden.

Im Anschluss muss spezifiziert werden, welche Daten gesammelt werden müssen, um die nötigen Informationen zu erhalten. Dabei kann sich herausstellen, dass für bestimmte Informationen die Datengrundlage fehlt oder neben dem Quellcode noch weitere Datenquellen miteinbezogen werden müssen. An dieser Stelle muss möglicherweise eine Abwägung zwischen Aufwand und Nutzen der Implementierung vorgenommen und die Auswahl der zu analysierenden Aspekte angepasst werden.

Da die Daten, die über eine Software gesammelt werden, ein Modell dieser Software darstellen, wird die Struktur der Daten selbst im Folgenden *Metamodell* genannt. Dieses Metamodell muss auf geeignete Weise beschrieben werden und dient als Grundlage für die anderen Teile der Arbeit.

Der nächste Schritt ist die Umsetzung der Datenextraktion, welche automatisiert stattfinden soll. Die Eingabe ist dabei eine zu analysierende Software. Das Ergebnis ist das oben erwähnte Modell dieser Software (Abbildung 1.1). Je nach Inhalt des Modells kann es sein, dass bestimmte Informationen nicht automatisiert gewonnen werden können, sondern durch den Benutzer zur Verfügung gestellt werden müssen. Solche manuellen Eingaben sollten aber soweit wie möglich vermieden werden, um eine einfache Benutzung zu garantieren.

Für die Implementierung der Datenextraktion muss eine geeignete Programmier- oder Skriptsprache ausgewählt werden. Dabei spielt auch die Verfügbarkeit externer Bibliotheken eine Rolle, die den Implementierungs- und Wartungsaufwand verrin-

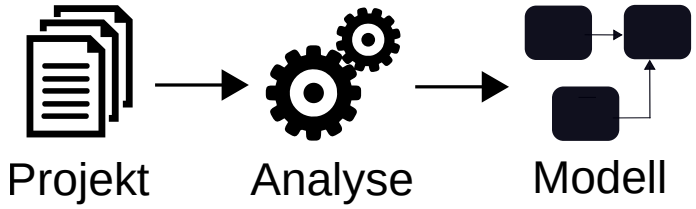


Abbildung 1.1: Die Analyse extrahiert Daten aus einem Softwareprojekt und erstellt daraus ein Modell dieser Software.[1]

gern können.

Nach der Datenextraktion und der Erstellung des Modells kann dieses visualisiert werden. Die Konzeption der Visualisierung stellt einen weiteren wichtigen Teil dieser Arbeit dar. Ausgehend von bisherigen Arbeiten zum Thema Softwarevisualisierung sollen Darstellungsarten ausgewählt werden, welche sich für die, zu Beginn ausgewählten Aspekte der Softwarearchitektur eignen und ihren Zweck hinsichtlich der definierten Ziele erfüllen.

Die Visualisierung soll nicht nur Metriken darstellen, sondern eine Exploration der Softwarearchitektur ermöglichen, die durch Betrachtung des Quellcodes und statischer Diagramme nicht möglich wäre. Dazu muss der Benutzer mit der Visualisierung interagieren und zum Beispiel Informationen filtern und selektieren können. Ob eine Aufteilung der Visualisierung auf mehrere, eventuell verknüpfte, Darstellungen bzw. „Views“ sinnvoll ist, muss währed der Konzeption entschieden werden.

Auch bei der Implementierung der Visualisierung müssen einige technische Entscheidungen getroffen werden. Neben der Wahl der Programmiersprache stellt sich auch die Frage, in welcher Umgebung die Visualisierung laufen soll und wie die Verbindung zwischen Datenanalyse und Visualisierung aussehen soll. Hier muss auch überlegt werden, wie die Visualisierung sinnvoll in vorhandene Software-Entwicklungsprozesse einbezogen werden könnte. Denkbar ist beispielsweise die Integration in ein serverseitiges Continuous-Integration-System oder eine clientseitige Entwicklungsumgebung.

Aus Zeitgründen muss sich die Implementierung auf einige Teile des Visualisierungskonzeptes beschränken. Eine umfassende Umsetzung ist im Rahmen der Masterarbeit nicht möglich. Der Fokus soll darauf liegen, das Potential und den praktischen Nutzen des Konzepts zu demonstrieren und die Grundlagen für weitere Entwicklungen zu legen.

Generell sollen alle entwickelten Komponenten, Metamodell, Datenanalyse und Visualisierung, so entworfen werden, dass eine spätere Erweiterung möglich ist.

1.3 Anwendungsbeispiel

Die Idee für diese Masterarbeit kam vom Deutschen Zentrum für Luft- und Raumfahrt. Dort wird in der Einrichtung „Simulations- und Softwaretechnik“ die sogenannte *Remote Component Environment* [87], kurz RCE, entwickelt.

[1] Bildquelle des File-Icons: [92]

RCE ist eine OSGi-Software und dient als Anwendungsbeispiel an dem die, in dieser Arbeit erstellte, Visualisierung evaluiert werden soll. Diese soll aber keineswegs auf RCE beschränkt sein, sondern auf möglichst alle Anwendungen anwendbar sein, die auf den offenen OSGi-Standard setzen.

1.4 Gliederung der Arbeit

In Kapitel 2 werden zunächst die nötigen Grundlagen für die Analyse von OSGi-Software beschrieben. Anschließend werden in Kapitel 3 theoretische Grundlagen der Datenvisualisierung und der Stand der Technik im Bereich der Softwarevisualisierung erläutert. Basierend darauf wird in Kapitel 4 ein Konzept für die Analyse und Visualisierung von OSGi-Anwendungen entworfen. In Kapitel 5 wird die Implementierung des Konzepts aufgeteilt auf drei Teile (Metamodellierung, Extraktion/Analyse und Visualisierung) vorgestellt. Anhand mehrerer User-Stories und am Beispiel von RCE wird in Kapitel 6 die Evaluierung der Implementierung beschrieben und ein Fazit gezogen. Abschließend werden die Ergebnisse in Kapitel 7 zusammengefasst und ein Ausblick auf mögliche Erweiterungen des Konzepts und der Implementierung gegeben.

2 Grundlagen

In diesem Kapitel werden einige Grundlagen dieser Arbeit vorgestellt. Zunächst werden die relevanten Konzepte von Java und OSGi erläutert. Danach wird auf einige Themen eingegangen, die im Hinblick auf die Analyse von OSGi-Software von Bedeutung sind.

2.1 Java

Java ist eine statisch typisierte, objektorientierte Programmiersprache, die universell, vor allem in der Anwendungsentwicklung, eingesetzt wird. Java-Quellcode wird vor der Ausführung in sogenannten *Bytecode* kompiliert. Dabei handelt es sich um eine binäre, plattformunabhängige, low-level Darstellung des Programms. Zur Laufzeit wird der Bytecode von der *Java Virtual Machine* (JVM) ausgeführt [22].

Zur Ausführung eines in Java geschriebenen Programms, muss also eine JVM auf dem Zielsystem vorhanden sein. Da Bytecode plattformunabhängig ist, bedeutet dies aber gleichzeitig, dass Java-Anwendungen ohne Änderungen am Quellcode und sogar ohne erneute Kompilierung auf verschiedener Hardware und unter unterschiedlichen Betriebssystemen ausgeführt werden kann, solange eine JVM für das System existiert. Diese Plattformunabhängigkeit ist eine zentrale Eigenschaft von Java und spiegelt sich auch in dessen Slogan „*write once, run anywhere*“ wieder.

2.1.1 Struktur

Java-Quellcode wird in Dateien mit der Endung `.java` gespeichert. Solch eine Datei wird auch als *Compilation Unit* bezeichnet, da dies die Einheit ist, die der Java-Compiler als Eingabe verarbeitet. Eine Compilation Unit enthält immer eine Typ-Deklaration, die eine Klasse definiert[1]. Eine Klasse wiederum kann beliebig viele weitere Typ-Deklarationen (auch *Nested Classes*) enthalten.

Neben der „normalen“ Klasse, wie sie beispielsweise auch in C++ vorkommt, gibt es in Java noch andere Arten von Klassen wie abstrakte Klassen, Interfaces, Enums und Annotationen [22].

Zur Modularisierung und Aufteilung des Quellcodes werden Klassen in Packages organisiert. Diese stellen einen hierarchischen Namensraum für Klassen bereit, um Namenskonflikte zu vermeiden. Als *vollständig qualifizierter Name* einer Klasse wird die eindeutige Kombination des Package- und des Klassennamens bezeichnet. Eine Klasse `Foo` im Package `com.example` trägt beispielsweise den qualifizierten Namen

[1] Die Begriffe *Typ* und *Klasse* werden im Folgenden synonym verwendet. Wenn nicht anders erwähnt, ist damit der Oberbegriff für alle Arten von Typen in Java gemeint.

`com.example.Foo`. Die Package-Struktur spiegelt sich auch in der Ordnerstruktur einer Java-Anwendung wider. Die CompilationUnit einer Klasse muss standardmäßig in einem Verzeichnis liegen, dessen Ebenen den Namensteilen des Packages entspricht (Beispiel: `com/example/Foo.java`).

Packages können zur Kapselung auf der Granularität von Klassen verwendet werden, da der *Access-Modifier* (`public`, `protected`, `private`) der enthaltenen Klassen ihre Sichtbarkeit für Klassen anderer Packages festlegt. Packages sind in Java somit ein zentrales Konzept für die Modularisierung von Code. Darauf, warum dieses Modulkonzept aber nicht immer ausreichend ist, wird in Abschnitt 2.2 eingegangen.

Kompilierte Java-Anwendungen werden üblicherweise in *JAR*-Archive zusammengefasst und in diesem Format weitergegeben. Ein JAR-Archive entspricht einem ZIP-Archiv, das ein Verzeichnis mit Metadaten (`META-INF`) beinhaltet. Darin kann unter anderem eine Datei namens `MANIFEST.MF` liegen, die Informationen über Packages und Erweiterungen enthält, aber auch für anwendungsspezifische Metadaten genutzt werden kann.

2.1.2 Classloading

Die JVM lädt, linkt und initialisiert Klassen zur Laufzeit. Das Laden übernimmt dabei der sogenannte *Classloader*, dessen Aufgabe es ist, den Bytecode einer Klasse zu finden, zu lesen und eine Repräsentation der Klasse im Arbeitsspeicher zu erzeugen [39]. Dieser Prozess wird angestoßen, wenn eine Klasse zum ersten Mal von einer anderen Klasse referenziert wird.[2]

Es gibt zwei Arten von Classloadern: System-Classloader und User-Classloader. Erstere werden von der JVM bereitgestellt und standardmäßig zum Laden aller Klassen benutzt. Es gibt mehrere Systemclassloader, die verschiedene Arten von Klassen laden und in einer Hierarchie zueinander stehen. Der genaue Ablauf des Classloadings ist an dieser Stelle aber nicht weiter von Bedeutung. Zusätzlich zu den System-Classloadern können auch anwendungsspezifische Classloader implementiert und benutzt werden [38]. Dadurch kann beispielsweise Bytecode geladen werden, der nicht als Datei im lokalen Dateisystem vorliegt, sondern von einem anderen Ort bezogen werden muss.

2.2 OSGi

OSGi ist die Spezifikation eines Java-Frameworks, dass zur Erstellung und Ausführung dynamischer, modularer Systeme benutzt wereden kann [47] und ein komponenten- und serviceorientiertes Entwicklungsmodell unterstützt. Das Framework verwaltet den Lebenszyklus von Modulen und deren Abhängigkeiten und stellt Mittel zum Suchen und Veröffentlichen von Services zur Verfügung [73]. Es gibt diverse Implementierungen der OSGi-Spezifikation, die auch als *OSGi-Container* bezeichnet werden. Ein bekannter OSGi-Container ist *Equinox*, welches unter anderem als Grundlage für die Eclipse IDE dient [91].

[2] Der Prozess des Classloading beginnt bei der initialen Klasse, welche die `main`-Methode enthält

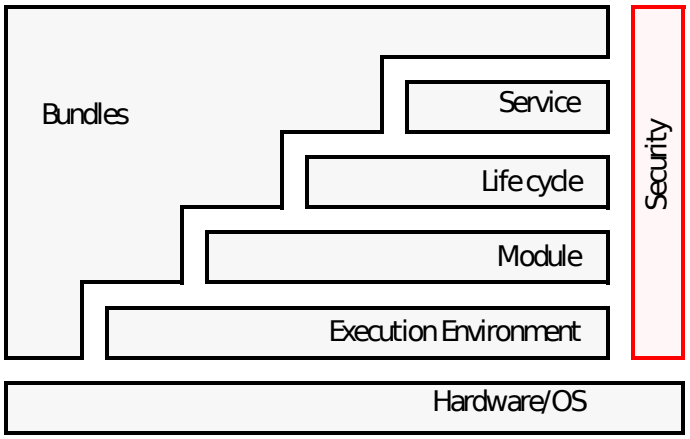


Abbildung 2.1: Aufbau des OSGi-Frameworks (Quelle: [3])

.

2.2.1 Framework

Das OSGi-Framework lässt sich in vier konzeptionelle Ebenen einteilen (siehe auch Abbildung 2.1):

- Module Layer
- Service Layer
- Lifecycle Layer
- Security Layer

Das Lifecycle Layer ermöglicht es, Module dynamisch in ein laufendes System zu Laden und dessen Abhängigkeiten zur Laufzeit aufzulösen. Der dynamische Aspekt von OSGi wird im Rahmen dieser Arbeit aber nicht berücksichtigt. Auch das Security Layer ist hier nicht relevant.

Module Layer

Wie schon in Abschnitt 2.1.1 angesprochen, sind Access Modifier und Packages Mittel, um Java-Software modular zu gestalten[3]. Die resultierende Modularität ist für große Anwendungen jedoch zu feingranular, da es zum Beispiel nicht möglich ist öffentliche Klassen eines Packages nur für bestimmte andere Packages freizugeben. Stattdessen sind alle öffentlichen Klassen global zugänglich. Auch die Aufteilung in JAR-Archive hilft hier nicht weiter. Bei diesen handelt es sich nur um ein Deployment-Hilfsmittel. Zur Laufzeit sind die Grenzen von JAR-Archiven aber nicht mehr von Bedeutung [78].

Aus diesem Grund beinhaltet OSGi ein übergeordnetes Modulkonzept, welches eine Kapselung auf Package-Ebene erlaubt. Module werden in OSGi als *Bundle* bezeichnet und in eigenen JAR-Archiven packetiert und verbreitet. Ein Bundle ist

[3]Eine Definition von wird in Abschnitt 2.3 gegeben.

Listing 2.1: Beispiel für ein Bundle-Manifest.

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: MyBundle
Bundle-SymbolicName: com.example.mybundle
Bundle-Version: 1.0.0
Bundle-Activator: com.example.mybundle.Activator
Import-Package: org.sample.util;version="1.0.4"
Export-Package: com.example.mybundle.api;version="1.0.0"
Require-Bundle: com.example.anotherbundle
```

eine abgeschlossene Einheit zusammengehörender Klasse, deren Abhängigkeiten zu anderen Modulen und Services explizit definiert sind. Außerdem kann ein Bundle selbst eine API auf Basis von Packages definieren. Dies hat zur Folge, dass nur die öffentlichen Klassen für andere Bundles sichtbar sind, die explizit exportiert wurden.

Informationen über ein Bundle und dessen API sind in der Manifest-Datei des zugehörigen JAR-Archivs enthalten. Diese kann wie in Listing 2.1 gezeigt aussehen. Die `Import-Package`-Anweisung gibt an, von welchen Packages das Bundle abhängt. Da Packages in OSGi versioniert sein können, kann beim Import eine minimal Versionsnummer mit angegeben werden. Von welchem anderen Bundle das Package `org.sample.util` bereitgestellt wird, ist an dieser Stelle nicht wichtig. Um die Auflösung der Abhängigkeiten kümmert sich der OSGi-Container zur Laufzeit. So ist eine geringe Kopplung zwischen Bundles möglich. Das Gegenstück zur `Import-Package`-Anweisung ist `Export-Package`, womit die API des Bundles definiert wird [3].

Ein Sonderfall ist `RequireBundle`. Mit dieser Anweisung werden implizit alle exportierten Packages eines anderen Bundles importiert. Dadurch ensteht eine feste Abhängigkeit zur einem bestimmten Bundle.

Service Layer

Das OSGi Service Layer ermöglicht eine zuverlässige Kommunikation zwischen Bundles über Interfaces, ohne, dass eine explizite Abhängigkeiten zwischen diesen nötig ist. Durch einen deklarativen Ansatz zum bereitstellen und benutzen von Services ist dazu nur wenig Quellcode auf Anwendungsseite nötig. Genau wie Bundles können Services zur Laufzeit hinzukommen oder wegfallen. Zudem werden Services erst dann geladen, wenn sie tatsächlich benötigt werden [2].

Zur Deklaration von Services werden sogenannte *Declarative Service Components* benutzt, die durch je eine XML-Datei definiert werden (siehe Listing 2.2). Ein Service-Component kann Services anbieten (`provide`) und Services verwenden (`reference`). Ein Service ist definiert durch ein Java-Interface. Jedes Service-Component braucht zudem eine Klasse, durch die es implementiert wird. Diese Klasse kann die referenzierten Services benutzen und muss Service-Interfaces von bereitgestellten Services implementieren.

Listing 2.2: Beispiel für die Deklaration eines Service-Components.

```
<scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0"
    name="com.example.MyComponent">
  <implementation class="com.example.impl.MyServiceImpl"/>
  <service>
    <provide interface="com.example.spi.MyService"/>
  </service>
  <reference interface="org.sample.AnotherService"
      cardinality="1..1"
      name="Service" policy="static"
      bind="bindService" unbind="unbindService"/>
</scr:component>
```

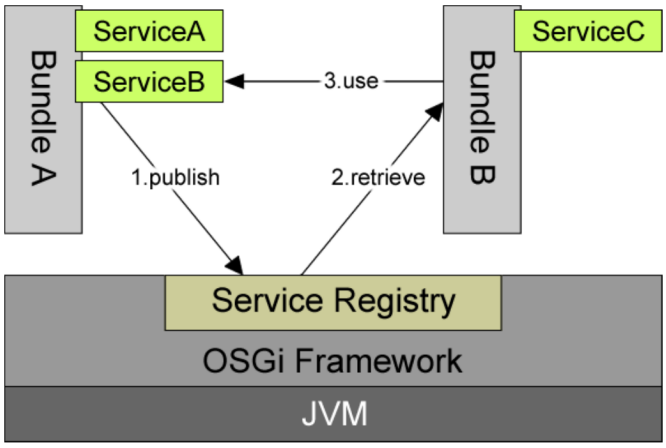


Abbildung 2.2: Bundles und Services im OSGi-Framework (Quelle: [73])

.

Jedes Bundle kann mehrere Service-Components besitzen und über diese Services benutzen und bereitstellen (siehe Abbildung 2.2).

2.2.2 Classloading

Viele Funktionen von OSGi wie zum Beispiel die Kapselung ganzer Packages in Modulen, werden erst durch einen speziellen Classloading-Mechanismus möglich. Jedes Bundle in einem OSGi-Container besitzt einen eigenen Classloader. Dieser wird von der OSGi-Implementierung bereitgestellt und ist dafür zuständig, die Klassen des Bundles zu laden [82]. Wird eine Klasse aus einem anderen Bundle benötigt, wird dazu der entsprechende Classloader angefragt.

Jeder Classloader (und somit jedes Bundle) verfügt über einen eigenen Namespace für Klassen. Dadurch ist es möglich, dass in einer Anwendung mehrere Klassen existieren können, deren vollständig qualifizierte Namen identisch sind. Dieser Umstand muss bei der Metamodellierung in Abschnitt 5.1 berücksichtigt werden.

2.3 Modularität

Die Modularität von Software spielt in dieser Arbeit eine wichtige Rolle, da ein Ziel von OSGi eine modulare Architektur ist und Beziehungen zwischen Modulen betrachtet werden sollen. Module gruppieren zusammengehörige Programmteile und stellen nach außen hin Services[4] für andere Module über eine definierte Schnittstelle zur Verfügung [70]. Dadurch wird unter anderem die Wiederverwendung von Code gefördert und die Komplexität bei der Entwicklung durch Zerlegung verringert [66].

Modularität kann auf verschiedenen Abstraktionsebenen im Code angewandt werden. Im weitesten Sinne sind Methoden, Klassen, Packages, Bundles und sogar ganze Anwendungen jeweils eine Art von Modul. In den folgenden Kapiteln wird das Wort „Komponente“ benutzt, wenn diese umfassende Bedeutung gemeint ist. Die Bezeichnung „Modul“ hingegen bezieht sich explizit auf Komponenten in der Abstraktionsebene von Bundles.

Modularität fasst im Prinzip zwei Attribute, nämlich *Kopplung* und *Kohäsion*, zusammen. Geringe Kopplung zwischen Modulen und hohe Kohäsion innerhalb von Modulen ist das Ideal, welches angestrebt wird [66].

2.3.1 Kopplung

Kopplung ist ein Maß dafür, wie stark zwei Komponenten zusammenhängen [63], wobei sich daraus auch eine globale Kopplung bestimmen lässt, die sich auf alle Komponenten bezieht [18]. Bei der Analyse von Kopplung ist zu beachten, dass diese auf verschiedene Arten zustande kommen kann. Einige Arten gelten als weniger kritisch als andere.

Beck und Diehl [6] liefern eine umfassende Liste von Möglichkeiten, durch die Kopplung bzw. Abhängigkeiten zwischen Klassen entstehen können. Die komplette

[4] An dieser Stelle sind Services im Allgemeinen und nicht speziell die Services in OSGi gemeint.

Auflistung kann an dieser Stelle nicht im Detail wiedergegeben werden. Hier aber eine Zusammenfassung der Kopplungsarten, die für die späteren Kapitel von Relevanz sind[5]:

- Strukturelle Kopplung:
  1. Eine Klasse ruft Methoden einer anderen Klasse auf.
     (*Benutzung*)
  2. Eine Klasse verwendet eine andere Klasse als lokale Variable oder Methodenparameter.
     (*Benutzung*)
  3. Eine Klasse deklariert Variablen vom Typ einer anderen Klasse.
     (*Aggregation*)
  4. Eine Klasse erbt direkt oder indirekt von einer anderen Klasse.
     (*Vererbung*)
  5. Zwei Klassen erweitern die gleiche Klasse oder implementieren die gleichen Schnittstellen.
     (*Fan-Out-Ähnlichkeit*)
- Evolutionäre Kopplung:
  6. Mehrere Klassen wurden während der Entwicklung zusammen geändert.
     (*Support*)
  7. Mehrere Klassen wurden vom gleichen (mehreren) Autor(en) verändert.
     (*Besitz*)
- Semantische Kopplung:
  8. Mehrere Klassen teilen das gleiche Vokabular in Variablenbezeichnungen und Kommentaren.
     (*Vokabular*)
  9. Mehrere Klassen sind Teil derselben Funktionalität des Programms.
     (*Funktion*)

In OSGi-basierten Anwendungen können nicht nur Abhängigkeiten zwischen Klassen, sondern auch zwischen Bundles betrachtet werden. Analog zu obiger Einteilung kann man dabei folgende Arten von Kopplung unterscheiden:

- Strukturelle Kopplung:
  1. Ein Bundle importiert Packages eines anderen Bundles.
     (*Import*)
  2. Ein Bundle deklariert ein anderes als explizite Abhängigkeit (das heißt Import aller Packages).
     (*Require*)

[5]Die kursiven Bezeichnungen in Klammern geben der Art der Kopplung einen Namen, sodass später wieder darauf Bezug genommen werden kann.

3. Ein Bundle benutzt einen Service, der von einem anderen Bundle bereitgestellt wird.
(*Service*)

- Evolutionäre Kopplung:
  4. Mehrere Bundles wurden während der Entwicklung zusammen geändert.
(*Support*)
  5. Mehrere Bundles wurden vom gleichen (mehreren) Autor(en) verändert.
(*Besitz*)

- Semantische Kopplung:
  6. Mehrere Bundles beinhalten Klassen im selben Package[6].
(*Split*)
  7. Mehrere Bundles sind Teil derselben Funktionalität des Programms.
(*Funktion*)

2.3.2 Kohäsion

Kohäsion ist ein Maß dafür, wie stark die Einzelteile einer Komponente zusammenhängen und wie sehr sie sich alle auf „eine Sache“ fokussieren („Single-Mindedness“) [63].

Auch für Kohäsion gibt es Klassifizierungsmöglichkeiten, ähnlich denen der Kopplung [18]. Da Kohäsion in dieser Arbeit aber nur eine untergeordnete Rolle spielt, wird hier nicht näher auf deren Arten eingegangen.

2.4 Softwareanalyse

Der Begriff der Softwareanalyse, oder auch Programmanalyse, wird vielseitig verwendet und in der Literatur zum Teil sehr unterschiedlich definiert. Grund dafür ist, dass es mehrere Arten von Softwareanalyse gibt und verschiedenste Ziele, die damit verfolgt werden.

Üblicherweise werden statische und dynamische Analyse unterschieden. Erstere untersucht ein Programm, meist in seiner Darstellungsform als Programmcode, losgelöst von seiner Ausführungsumgebung, das heißt zur „Compile-Zeit“. Eine Ausführung des Programms ist dazu also nicht nötig [76]. Eine klassische Anwendung von statischer Analyse sind Compiler, mit dem Ziel den Code auf Korrektheit zu überprüfen oder Optimierungsmöglichkeiten zu erkennen.

Bei der dynamischen Analyse hingegen wird das Verhalten der Software während ihrer Ausführung beobachtet. Die Ziele können, ähnlich der statischen Analyse, Optimierung, zum Beispiel durch Erkennung von Performance-Engpässen, sowie Korrektheit, zum Beispiel durch Tests, sein.

Das Ziel dieser Arbeit ist weder die Korrektheitsprüfung noch die Optimierung von Code, sondern dessen Visualisierung. Wenn im Folgenden von Softwareanalyse die

[6] „Split-Packages“ (Abschnitt 2.2)

Rede ist, dann ist damit die Gewinnung von Daten aus der Software gemeint, welche von der Visualisierung genutzt werden können. Obige Unterscheidung von statischer und dynamischer Analyse gilt weiterhin. Der Gegenstand der Analyse wird aber vom reinen Programmcode auf alles erweitert, was ein Softwareprojekt ausmacht. Das sind nicht nur Repräsentationen des Programmcodes, sondern beispielsweise auch Metadaten über den Entwicklungsprozess.

2.5 Software-Measurement

Auf die aus der Analyse gewonnenen Daten können Funktionen angewandt werden, um quantitative Werte zu erhalten, die Maße oder Bewertungen bestimmter Aspekte der Software darstellen. Dieser Prozess wird „Software-Measurement“, zu Deutsch „Softwaremessung oder -bewertung“, genannt. Sowohl besagte Funktionen als auch die resultierenden Werte werden im Folgenden als Metriken bezeichnet.

Visuelle Darstellungen von Software können, auch wenn sie sich eher auf Strukturen und Beziehungen konzentrieren, durch Metriken angereichert werden und dem Betracher so viele zusätzliche Informationen präsentieren. Dazu müssen die Metriken auf grafische Variablen abgebildet werden. Beispielsweise könnte eine Metrik für die Größe von Klassen in ein Klassendiagramm einbezogen werden, indem die Größe der Rechtecke in der Darstellung entsprechend der Metrik verändert wird.

Die nachfolgenden Erläuterungen orientieren sich an den Definitionen von Fenton [18]. Seine allgemeine Definition des Begriffs „Measurement“ lautet:

> *„Measurement is concerned with capturing information about attributes of entities. (...) Measurement assigns numbers or symbols to these attributes of entities in order to describe them.“*

Bezogen auf Software gibt es drei Arten von Entitäten:

Produkte: Dokumente und andere Artefakte, die während der Entwicklung entstehen. Darunter fallen der Quellcode, aber auch Spezifikations- und Designdokumente, sowie Testdaten.

Prozesse: Aktivitäten, durch welche die Produkte entstehen. Also zum Beispiel Konzeption und Programmierung.

Ressourcen: Dinge, die einen Beitrag zu Prozessen leisten. Dies können Personal, Materialien, Werkzeuge und Methoden sein.

Metriken werden auf Attribute dieser Entitäten angewandt. Man kann zwischen internen und externen Attributen unterscheiden.

Zur Bewertung interne Attribute müssen nur die jeweiligen Produkte, Prozesse oder Ressourcen selbst betrachtet werden. Im Fall von Quellcode sind interne Attribute: Größe, Modularität, algorithmische Komplexität, Strukturiertheit des Kontrollflusses, Änderungshäufigkeit, etc. Interne Attribute können meistens *direkt* gemessen werden, das heißt ohne auf andere Attribute zurückgreifen zu müssen.

Während sich Softwareentwickler hauptsächlich mit internen Attributen befassen, sind externe Attribute vor allem für Manager und Benutzer interessant.

Externe Attribute können oft nur indirekt, durch die Betrachtung von internen Attributen, bestimmt werden. Daraus folgt, dass erstere von letzteren beeinflusst werden. Beispiele für externe Attribute sind: Zuverlässigkeit, Wartbarkeit, Kosteneffizienz und Qualität im Allgemeinen. In Tabelle 2.1 werden einige Entitäten und Attribute des Software-Measurements beispielhaft eingeordnet.

Im Rahmen dieser Arbeit werden ausschließlich interne Attribute betrachtet. Der Benutzer der Visualisierung sollte daraus aber Rückschlüsse auf externe Attribute der Software ziehen können (vgl. Abschnitt 4.2). Beispielsweise könnte aus dem Maß der Kopplung zwischen Modulen (internes Attribut) eine Aussage über die Wartbarkeit der Software (externes Attribut) getroffen werden.

2.6 Metriken

Es gibt eine unüberschaubare Anzahl an Metriken, die beim Software-Measurement zum Einsatz kommen können. Im Folgenden soll nur eine Übersicht über einige gängige Metriken gegeben werden, die sich auf in dieser Arbeit relevante Attribute (Größe, Kopplung, Köhäsion) anwenden lassen.

2.6.1 Größe

Das Attribut *Größe* kann im Bezug auf Software verschiedene Bedeutungen haben. Zum Beispiel kann man zwischen funktionaler und technischer Größe unterscheiden [1]. Erstere ist die Größe von Komponenten aus Benutzersicht und stellt eine quantitative Bewertung der Funktionalität dar. Diese ist unabhängig von der Implementierung und der verwendeten Programmiersprache. Im Gegensatz dazu wird die technische Größe in implementierungsspezifischen Metriken gemessen, wie „Lines of Code“ (LOC) oder Anzahl an Klassen. Eine weitere Möglichkeit, die Größe von Software zu bestimmen, ist die Betrachtung der Komplexität des zugrunde liegenden Problems bzw. von Algorithmen.

Laut Fenton [18] sollte die Größe einer Software idealerweise aus der Kombination dieser drei orthogonalen Attribute (technische Größe, Funktionalität, Komplexität) bestimmt werden.

Funktionale Größe

Ein verbreitetes Verfahren zur Bestimmung der funktionellen Größe ist die *Function-Point-Analyse* (FPA), die als ISO-Norm ISO/IEC 20926 standardisiert wurde und auf deren Grundlage diverse weitere Metriken entwickelt wurden [1]. Da solche Methoden aber vorwiegend der Aufwandsschätzung in Softwareprojekten dienen, sind sie an dieser Stelle nicht weiter von Interesse.

Entitäten	**Attribute**	
Produkte	**Intern**	**Extern**
Spezifikationen	Größe, Wiederverwendung, Modularität, Redundanz, Funktionalität, Syntaktische Korrektheit ...	Verständlichkeit, Wartbarkeit ...
Design	Größe, Wiederverwendung, Modularität, Kopplung, Kohäsion, Funktionalität ...	Qualität, Komplexität, Wartbarkeit ...
Code	Größe, Wiederverwendung, Modularität, Kopplung, Funktionalität, algorithmische Komplexität, Strukturiertheit des Kontrollflusses ...	Zuverlässigkeit, Benutzbarkeit, Wartbarkeit ...
Testdaten	Größe, Testabdeckung ...	Qualität ...
...		
Prozesse	**Intern**	**Extern**
Spezifikation	Zeit, Aufwand, Anzahl von Anforderungsänderungen ...	Qualität, Kosten, Stabilität ...
Design	Zeit, Aufwand ...	Kosteneffizienz, Kosten ...
Tests	Zeit, Aufwand, Anzahl gefundener Bugs ...	Kosteneffizienz, Stabilität, Kosten ...
...		
Ressourcen	**Intern**	**Extern**
Personal	Alter, Gehalt ...	Produktivität, Erfahrung, Intelligenz ...
Teams	Größe, Kommunikationslevel, Organisiertheit ...	Produktivität, Qualität ...
Software	Preis, Größe ...	Benutzbarkeit, Zuverlässigkeit ...
Hardware	Preis, Geschwindigkeit, Speicheranforderungen ...	Zuverlässigkeit ...
Büros	Größe, Temperatur, Lichtverhältnisse ...	Komfort, Qualität ...
...		

Tabelle 2.1: Verschiedene Arten von Entitäten und Attributen beim Software-Measurement (nach Fenton [18]).

Technische Größe

In der, in Kapitel 5 beschriebenen Implementierung, werden eher einfach umzusetzende Metriken verwendet, die sich auf die technische Größe von Komponenten beschränken.

Triviale Kennzahlen für die Größe von Klassen sind zum Beispiel die Anzahl von Code-Zeilen (LOC), Attributen (das heißt Member-Variablen) oder Methoden. Doch auch diese Metriken lassen sich nach Bedarf verfeinern. So können etwa verschiedene Arten von Methoden (abstrakte, statische, öffentliche, private ...) unterschieden und anders bewertet werden.

Beim Zählen von Code-Zeilen bietet es sich an, Kommentare und Leerzeilen zu ignorieren. In diesem Fall spricht man von *physikalischen Code-Zeilen*, welche unabhängig von der verwendeten Programmiersprache sind. Eine Alternative ist die Betrachtung von *logischen Code-Zeilen*, bei denen einzelne Instruktionen gezählt werden [52]. Diese werden in vielen Programmiersprachen durch ein Semikolon abgeschlossen und können mehrfach in einer physikalischen Code-Zeile vorkommen oder sich über mehrere erstrecken.

Schon solche einfachen Kennzahlen können helfen, Aussagen über Entwicklungs- und Wartungsaufwand einzelner Klassen zu machen. Zum Beispiel sind Klassen mit einer großen Methodenzahl wahrscheinlich relativ anwendungsspezifisch, was die Wiedeverwendbarkeit einschränkt. Zudem ist der Einfluss solcher Klassen auf das Verhalten von Unterklassen durch die Vererbung groß, was sich negativ auf den Wartungsaufwand auswirken kann [14].

Einfache Kennzahlen lassen sich auch zu weiteren Metriken kombinieren. Die Metrik *Weighted Methods per Class* (WMC) lässt sich zum Beispiel wie folgt berechnen:

$$\mathrm{WMC} = \sum_i c_i \tag{2.1}$$

wobei c_i die Größe oder Komplexität der i-ten Methode der Klasse ist. Wie diese bestimmt wird ist dabei frei wählbar.

Bisher wurden nur Metriken beschrieben, die auf der Ebene von Methoden und Klassen anwendbar sind. Darüber liegende Entitäten, wie Packages und Bundles, stellen vor allem eine Gruppierung von Klassen dar. Um ihre Größe zu messen, bietet es sich deshalb an die Kennzahlen der enthaltenen Klassen zu aggregieren.

Hamza et al. [25] schlagen für OSGi-Anwendungen außerdem vor, Metriken zu benutzen, deren kleinste Granularität die Klasse ist, das heißt, für deren Bestimmung eine Analyse des Quellcodes gar nicht nötig ist. Bezogen auf Bundles wäre das:

- Anzahl der enthaltenen Packages
- Anzahl der enthaltenen Klassen
- Anzahl der enthaltenen abstrakten Klassen und Interfaces

Die Unterscheidung zwischen Klassen im Allgemeinen und abstrakten Klassen/Interface ist sinnvoll, da letztere explizit in anderen Metriken Verwendung finden (siehe Abschnitt 2.6.4).

Kopplungsart		Bewertung
Strukturell	Benutzung	Anzahl der benutzten Methoden der anderen Klasse
	Aggregation	Anzahl der benutzten Variablen vom Typ der anderen Klasse
	Vererbung	konstant: 1
	Fan-Out-Ähnlichkeit	Kosinus-Ähnlichkeit der Merkmalsvektoren der Vererbung
Evolutionär	Support	Anzahl der gemeinsamen Änderungen
	Besitz	Kosinus-Ähnlichkeit der Merkmalsvektoren der Autoren
Semantisch	Vokabular	Kosinus-Ähnlichkeit der Tf-idf-Vektoren[7]

Tabelle 2.2: Metrik für die Stärke von Kopplung zwischen Klassen basierend auf [25]. Für eine Erläuterung der genannten Kopplungsarten siehe Abschnitt 2.3.1.

Komplexität

Die Grenze zwischen Metriken für Komplexität und für technische Größe ist fließend. Algorithmische Komplexität im Sinne der theoretischen Informatik lässt sich durch statische Analyse zwar nicht bestimmen, trotzdem gibt es diverse Metriken, die auf Komplexitätsmessung abzielen.

Die *zyklomatische Komplexität* (auch *McCabe-Metrik*) zum Beispiel beschreibt die Komplexität einer Komponente mit der Anzahl der linear unabhängigen Pfade auf dem Kontrollflussgraphen [5]. Andere Verfahren betrachten den Informationsfluss zwischen Komponenten (Fan-In und Fan-out) zur Bestimmung der Komplexität der einzelnen Komponenten. Die sogenannte *Halstead-Metrik* hingegen benutzt die Anzahl an Operatoren und Operanden, um verschiedene Kennzahlen zu berechnen [1].

2.6.2 Kopplung

Da in modularem Design geringe Kopplung im Allgemeinen als positive Eigenschaft angesehen wird, möchte man bei dessen Analyse auch Metriken für die Stärke von Kopplung einsetzen. Beck und Diehl [6] machen in ihrer Arbeit Vorschläge, wie die verschiedenen Arten von Kopplungen zwischen zwei Klassen, wie sie in Abschnitt 2.3.1 erläutert wurden, bewertet werden können. Tabelle 2.2 zeigt eine Übersicht dazu.

[7] Mit dem *Tf-idf*-Maß (*term frequency - inverse document frequency*) kann bestimmt werden, welche Relevanz ein Term (hier zum Beispiel ein Variablen-Bezeichner) in einem Dokument (hier Quellcode einer Klasse) hat [64]. Daraus lässt sich für jede Klasse ein Merkmalsvektor,

Diese Metrik lässt sich, wie in Tabelle 2.3 gezeigt, auf die Kopplung zwischen Bundles übertragen.

Kopplungsart		**Bewertung**
Strukturell	Import	Anzahl der importierten Packages des anderen Bundles
	Require	Anzahl der Bundles
	Service	Anzahl der benutzten Services
Evolutionär	Support	Anzahl der gemeinsamen Änderungen
	Besitz	Kosinus-Ähnlichkeit der Merkmalsvektoren der Autoren
Semantisch	Vokabular	Anzahl der Split-Packages

Tabelle 2.3: Metrik für die Stärke von Kopplung zwischen Bundles. Für eine Erläuterung der genannten Kopplungsarten siehe Abschnitt 2.3.1.

Dabei wird die Anzahl an importierten Packages als Bewertung für strukturelle Kopplung verwendet. Da das Importieren eines Packages aber nicht bedeutet, dass alle Klassen aus dem Package tatsächlich benutzt werden, könnte man die Metrik an dieser Stelle noch feingranularer gestalten, indem die Anzahl der tatsächlich referenzierten Klassen als Kennzahl verwendet wird. Das Gleiche gilt für die Kopplung durch explizit benötigte Bundles.

Während die bisherigen Metriken sich auf die Kopplung zwischen einer bestimmten Komponente und einer oder mehrerer anderer Komponenten bezieht, gibt es auch Ansätze die Kopplung global, also zwischen beliebig vielen Komponenten, zu messen.

Angenommen CD_i ist die Anzahl der Klassen von denen Klasse i direkt oder indirekt abhängt und Nc ist die Anzahl aller Klassen, dann kann die globale Kopplung berechnet werden durch [34]:

$$ACD = \frac{1}{Nc} \sum_{i=1}^{Nc} CD_i \tag{2.2}$$

Wohlgemerkt ist diese Variante sehr simpel, da hier keinerlei Unterscheidung nach der Art der Abhängigkeit gemacht wird.

2.6.3 Kohäsion

Im Vergleich zu Kopplung ist es bei Kohäsion nicht ganz so offensichtlich, wie diese gemessen werden kann. Im Folgenden werden zwei Kohäsionsmetriken, eine für Klassen und eine für Bundles, vorgestellt.

der die Relevanz aller Bezeichner für dieses Dokument enthält, aufstellen. Die Ähnlichkeit dieser Vektoren liefert ein Maß für die semantische Ähnlichkeit der Klassen.

Die Kohäsion einer Klasse C kann daran festgemacht werden, wie viel deren Methoden $M_1, \ldots, M_n$ gemeinsam haben. Seien M_1 und M_2 zwei Methoden von C und I_1 und I_2 die Mengen der Instanzvariablen auf die sie zugreifen. Dann ist die Gemeinsamkeit von M_1 und M_2 gegeben durch [14]:

$$\sigma_{1,2} = I_1 \cap I_2 \tag{2.3}$$

Über die paarweise Berechnung der Gemeinsamkeit für alle Methoden der Klasse C können die beiden Mengen $P = \{(M_i, M_j) | \sigma_{i,j} = \emptyset\}$ und $Q = \{(M_i, M_j) | \sigma_{i,j} \neq \emptyset\}$ gebildet werden. Daraus ergibt sich die Kohäsion der Klasse C durch:

$$\text{LOCM} = \begin{cases} |P| - |Q| & \text{, wenn } |P| > |Q| \\ 0 & \text{, sonst} \end{cases} \tag{2.4}$$

LOCM steht dabei für *Lack of Cohesion in Methods* und ist die Anzahl der Methodenpaare, deren Ähnlichkeitsmenge leer ist, minus der Anzahl der Methodenpaare, deren Ähnlichkeitsmenge nicht leer ist.

Idee hinter dieser Metrik ist die Annahme, dass zwei Methoden, die nicht die selben Instanzvariablen benutzen, nichts gemeinsam haben und deshalb vermutlich nicht in eine gemeinsame Klasse gehören. Eine Klasse mit einem hohen LOCM sollte möglicherweise in mehrere Klassen aufgespalten werden. Dadurch werden objektorientierte Design-Prinzipien, wie Kapselung und „Separation of Concerns“, gefördert.

Für die Kohäsion von Bundles kann die sogennante *Relational-Cohesion*-Metrik eingesetzt werden. Sie misst, wie stark die Klassen des Bundles untereinander im Durchschnitt zusammenhängen:

$$\text{RC} = \frac{TD}{Nc} \tag{2.5}$$

wobei TD die absolute Anzahl an ein- und ausgehenden Abhängigkeiten zwischen allen Klassen und Nc die Anzahl aller Klassen ist [37]. Da die Abhängigkeiten zwischen den Klassen vermutlich eng mit der Kohäsion des Bundles zusammenhängen, liefert diese Metrik einen guten Anhaltspunkt für diese [25].

2.6.4 Sonstiges

Zusätzlich zu Größe, Kopplung und Kohäsion gibt es in komponentenbasierter Software noch andere Attribute, die durch Metriken erfasst werden können. Beispielsweise bietet es sich an, die öffentliche Schnittstelle, die von einem Bundle zur Verfügung gestellt wird, zu analysieren. Diese wird in OSGi durch die exportierten Packages explizit definiert.

Neben der Anzahl an exportierten Packages, kann auch die Abstraktheit des Interfaces bestimmt werden, wie sie von Martin [43] für objekt-orientierte Software vorgeschlagen wurde. Abstraktheit ist von Interesse, da es als gutes Design angesehen wird, wenn eine Komponente nur abstrakte Klassen (bzw. Interfaces) exportiert. Die Abstraktheit einer Bundle-Schnittstelle kann wie folgt definiert werden [25]:

$$\text{Abs} = \frac{1}{n} \sum_{i=1}^{n} \frac{Na(i)}{Nc(i)} \tag{2.6}$$

wobei n die Anzahl der exportierten Packages ist. Na und Nc sind die Anzahl der abstrakten Klassen/Interfaces bzw. aller Klassen des jeweiligen Packages.

3 Daten- und Softwarevisualisierung

In diesem Kapitel werden die nötigen Grundlagen der allgemeinen Datenvisualisierung und ihrer speziellen Ausprägung, der Softwarevisualisierung, erläutert.

3.1 Interaktive Datenvisualisierung

Datenvisualisierung ist ein Begriff der viele Teilbereiche umfasst (wissenschaftliche Visualisierung, Informationsvisualisierung, medizinische Visualisierung...) und nicht eindeutig definiert ist. Williams et al. definieren Datenvisualisierung wie folgt:

> *"In computer and information science it is [...] the visual representation of a domain space using graphics, images, animated sequences, and sound augmentation to present the data, structure, and dynamic behavior of large, complex data sets that represent systems, events, processes, objects, and concepts."*[30]

Ziel ist es, dem Benutzer einen Einblick in diese Systeme, Prozesse usw. zu ermöglichen [74] und ihn damit bei der Gewinnung von Erkenntnissen und dem Treffen von Entscheidungen zu unterstützen [79].

Visualisierung ist ein wichtiges Instrument, um der Informationsflut Herr zu werden und Informationen auf effiziente und effektive Art zu kommunizieren. Der Mensch ist darauf ausgerichtet über visuelle Wahrnehmung eine große Menge an Informationen schnell verarbeiten zu können. Während die Schnelligkeit beim Lesen von Text durch den sequentiellen Leseprozess begrenzt ist, können Bilder im menschlichen Wahrnehmungssystem parallelisiert verarbeitet werden [79].

3.1.1 Stufen der Datenvisualisierung

Um von den rohen Daten zu einer interaktiven Visualisierung zu gelangen, muss ein Prozess mit mehreren Stufen durchlaufen werden. Dieser ist in Abbildung 3.1 dargestellt und enthält folgenden Schritte [19]:

Acquire Laden der Daten, zum Beispiel aus einer Datei oder einer Datenbank.

Parse Parsen der Daten und Erstellen einer Datenstruktur.

Filter Daten, die nicht von Interesse sind entfernen.

Mine Anwenden von Methoden der Statistik oder des Data Mining, um Muster und Strukturen in den Daten zu erkennen.

Abbildung 3.1: Stufen der Datenvisualisierung

Represent Auswahl einer visuellen Darstellung, auf die die Daten abgebildet werden.

Refine Verbesserung der Darstellung durch Anpassung visueller Parameter.

Interact Hinzufügen von Interaktionsmöglichkeiten für den Benutzer zur Manipulation der Darstellung.

In der Praxis können, je nach Anwendung, einzelne Schritte entfallen oder auch iterativ wiederholt werden.

3.1.2 Grafische Semiologie

Semiologie ist die Lehre, die sich damit beschäftigt, wie Zeichen und Symbole auf eine Bedeutung abgebildet werden und welche Konventionen und Systeme zur Organisation es dabei gibt [50]. Während sich viele semiologische Systeme auf sprachliche Symbole konzentrieren, entwickelte Bertin die sogenannte *grafische Semiologie* [7].

Dabei handelt es sich um eine in sich geschlossene, systematische und umfassende Beschreibung der Aspekte der Datenvisualisierung. Sie erklärt zum einen welche grafischen Darstellungen es gibt und wie diese aufgebaut sind und stellt Regeln, zur Konstruktion und Anwendungen von grafischen Darstellungen vor.

Damit bietet die grafische Semiologie ein System, das dabei hilft, prägnante und leicht verständliche Grafiken zu erstellen und ist für die Schritte *Represent* und *Refine* des obigen Prozesses relevant. Bertins System ist bei weitem zu komplex um es hier komplett darzustellen. Einige seiner Definitionen sind aber bei den Erklärungen der in Kapitel 5 beschriebenen Visualisierung nützlich und helfen dabei, die Entscheidungen für bestimmte Darstellungsarten nachzuvollziehen. Deshalb wird an dieser Stelle kurz auf einige Grundlagen der grafischen Semiologie eingegangen.

Information

Bertin definiert eine *Information* I als eine Relation von Informationskomponenten K_n:

$$I \subseteq K_1 \times K_2 \times \ldots \times K_n \tag{3.1}$$

Informationen werden im Folgenden auch einfach als *Daten* bezeichnet, wobei die Anzahl der enthaltenen Komponenten auch als *Dimension* der Daten betrachtet werden kann.

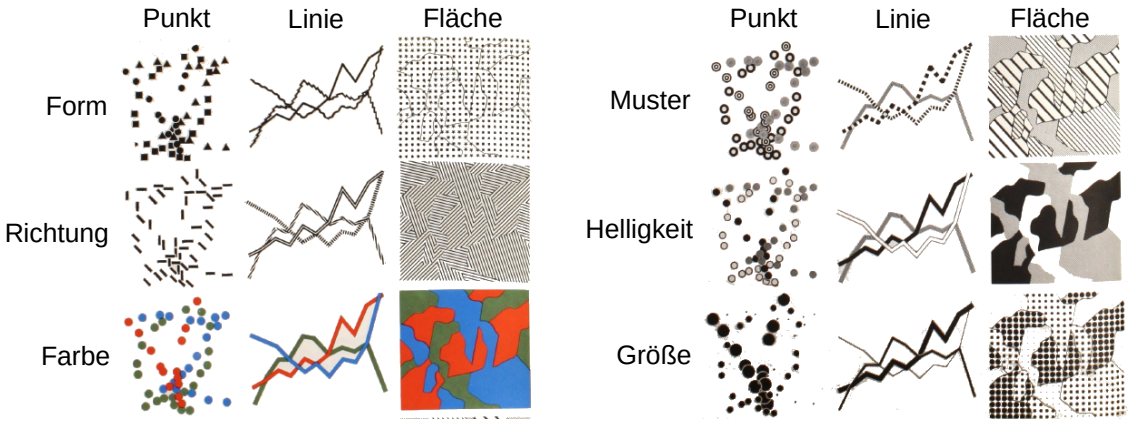


Abbildung 3.2: Grafische Variablen angewandt auf die elementaren Formen (nach [7])

Grafische Variablen

Um eine Information visuell darzustellen, müssen möglichst alle ihre Komponenten auf *grafische Variablen* abgebildet werden. Grafische Variablen sind: Position, Größe, Helligkeit, Textur, Farbe, Richtung und Form. In Abbildung 3.2 sind Beispiele für die Anwendung der grafischen Variablen auf die grundlegenden Formen (Punkt, Linie, Fläche) zu sehen.

Je nach Art der Informationskomponente, eignen sich bestimmte grafische Variablen aus wahrnehmungspsychologischer Sicht eher als andere. Stehen die Elemente der Komponente in keiner allgemeingültigen Ordnungsrelation zueinander oder stellen sie Kategorien dar, dann können Form oder Textur zu ihrer Unterscheidung dienen (Beispiel: Java-Klassen). Haben die Elemente aber eine Ordnung, beispielsweise eine Größenbeziehung und möchte man diese Ordnung visuell hervorheben, eignen sich dazu eher Helligkeit oder Größe (Beispiel: Lines of Code). Neben der Ordnung können grafische Variablen auch zur Gruppierung bzw. Trennung von Elementen eingesetzt werden.

Darstellungsarten

Bertin unterscheidet (beschränkt auf zwei Dimensionen) zwischen vier grundlegenden Darstellungsarten:

1. Diagramme
2. Netze
3. Karten
4. Symbole

Abbildung 3.3 zeigt einige Beispiele für diese Darstellungen.

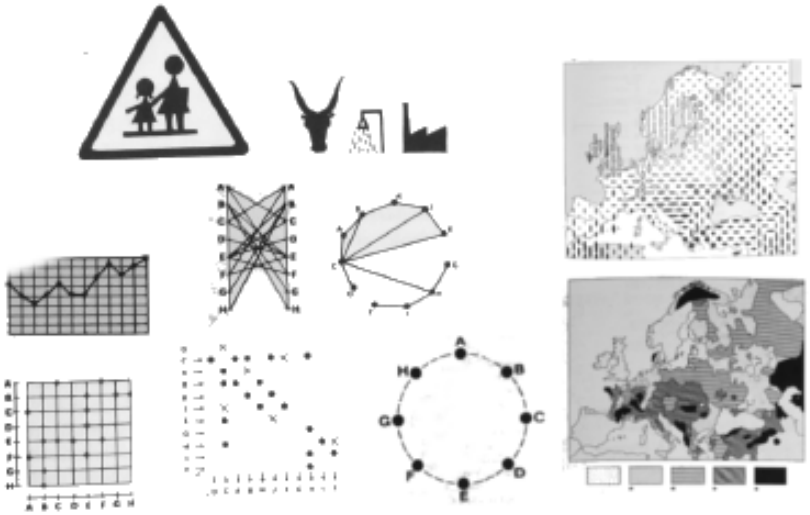


Abbildung 3.3: Beispiele für die vier Darstellungsarten: Diagramme, Netze, Karten und Symbole (nach [7]).

Unter Diagramme fallen die klassischen Diagrammarten, wie Balkendiagramm, Kreisdiagramm oder Scatterplots. Sie stellen häufig Beziehungen zwischen den Elementen von zwei Informationskomponenten dar (Beispiel: Lines of Code aufgetragen über die Zeit).

Netze sind vor allem Graph-Darstellungen, wie sie in der Informatik häufig vorkommen. Sie zeigen die Beziehungen zwischen den Elementen einer Informationskomponente (Beispiel: Abhängigkeiten zwischen Klassen).

Karten stellen Beziehungen zwischen Elementen von geografischen Komponenten dar und Symbole sind bildhafte Darstellungen, die keine Beziehungen zwischen Komponenten enthalten.

3.1.3 Interaktion und Exploration

Der letzte Schritt der Datenvisualisierung ist die Interaktion. Sie ermöglicht es dem Benutzer, die Darstellung an seine Bedürfnisse anzupassen und auszuwählen, *was* er *wie* zusehen bekommt. Folgende Interaktionsmethoden lassen sich unterscheiden [83, 79]:

Navigation Rotieren, Verschieben und Zoomen, sowie Wechseln verschiedener Ansichten (*Views*).

Filterung Reduzieren der angezeigten Daten.

Selektion Auswählen von Elementen, die von Interesse sind.

Rekonfigurierung Ändern des Layouts oder der Abbildung von Daten auf grafische Elemente.

Kodierung Anpassen von grafischen Variablen.

Verbindung Beziehungen zwischen verschiedenen Ansichten oder Objekten anzeigen.

Abstraktion/Vefeinerung Anpassung des *Level of Detail*.

Ein weiteres Konzept, dass mit Interaktion zusammhängt und im Rahmen dieser Arbeit von Bedeutung ist, ist die *Exploration*. Durch die Kombination von Datenvisualisierung und den genannten Interaktionsarten ist eine visuelle Exploration der Daten möglich. Durch diese kann der Benutzer sich ein mentales Modell der Daten aufbauen, neues Wissen entdecken und Strukturen, Muster, Beziehungen etc. identifizieren [79].

3.2 Softwarevisualisierung

Wie schon in der Einleitung erwähnt, ist Softwarearchitektur ein nicht direkt sichtbares, schwer fassbares Konzept, dass durch grafische Darstellungen und visuelle Metaphern verständlicher gemacht werden kann [62].

Nachweislich wird in der Wartungsphase einer Software (*Maintainance*) am meisten Zeit darauf verwendet, die Software zu verstehen [44]. Wenn Softwarevisualisierung das Verständnis fördert und so den teuren, zeitaufwändigen Prozess der Wartung verkürzt, kann sie effektiv einen Beitrag zur Reduzierung von Kosten in Softwareentwicklungsprojekten leisten [72, 11].

3.2.1 Definition und Ziele

Softwarevisualisierung, die sich auf die Präsentation der Software aus Sicht des Software-Engineering konzentriert, kann abgegrenzt werden von *Information Visualization* und *Program Visualization*. Bei der Information Visualization werden große Mengen abstrakter Daten (*Big Data*), die meistens durch Software erzeugt werden, visualisiert und die Program Visualization beschränkt sich auf die grafische Darstellung von Algorithmen [62]. Softwarevisualisierung hingegen soll einen Einblick in ein Softwaresystem bieten. Sie soll das Verständnis über das System unterstützen und dessen Komplexität reduzieren [33].

Die Aktivitäten eines Softwareentwicklers, die durch Visualisierung unterstützt werden können, lassen sich wie folgt einteilen [60]:

1. Verständnis (besonders von fremdem Code)
2. Debugging
3. Design-Entscheidungen

Je nach Aktivität sind eher low-level Aspekte (für Verständnis und Debugging) oder high-level Aspekte (für Design-Entscheidungen) bei der Visualisierung interessant.

Maletic et al. [40] beschreiben fünf Dimensionen der Softwarevisualisierung, aus denen sich Fragestellungen ableiten, die bei der Konzeption von Visualisierungen als Leitlinie dienen können:

Aufgabe *Warum* wird eine Visualisierung benötigt?

Zielgruppe *Wer* soll die Visualisierung benutzen?

Zielobjekt *Was* ist die Datenquelle, die dargestellt wird?

Darstellung *Wie* sieht die Darstellung aus?

Medium *Wo* wird die Visualisierung dargestellt?

Für die, in dieser Arbeit entwickelte Visualisierung, werden diese Fragen in Kapitel 4 beantwortet.

3.2.2 Herausforderungen

Die Menge an wissenschaftlichen Veröffentlichungen im Bereich der Softwarevisualisierung [32, 21, 11] legt nahe, dass es sich dabei um kein triviales Problem handelt.

Die Hauptherausforderung besteht darin, effektive Abbildungen von Softwareaspekten auf grafische Darstellungen durch visuelle Metaphern zu finden [23]. Das einfache Verpacken von großen textuellen Informationen in monumentale grafische Repräsentationen ist aber nicht zielführend [60].

Einige klassische Formen der Visualisierung, wie zum Beispiel der Call-Graph, bieten zwar Einblicke in das System, schaffen es aber nicht, die vom Benutzer wahrgenommene Komplexität zu reduzieren [33]. Ein Ansatz für die Lösung dieses Problems sind die oben vorgestellten Interaktionstechniken wie Filterung, Skalierung und Selektion [41].

Neben dem Problemen der Skalierbarkeit und der Vermeidung von Informationsüberflutung gibt es noch andere Herausforderungen denen bei der Visualisierung von Software begegnet werden muss.

Eine davon ist die Verknüpfung verschiedener Darstellungen (im Folgenden auch als *Views* bezeichnet). Durch das Erstellen einer Visualisierung sind schon mindestens zwei Views vorhanden: der Quelltext und die grafische Darstellung. Oft ist es sogar sinnvoll, mehrere grafische Darstellungen zu entwickeln, um mehr Informationen vermitteln zu können und dem Benutzer mehr Raum zur Exploration zu geben. Außerdem lassen sich so verschiedene Abstraktionsebenen darstellen und spezielle Visualisierungen für bestimmte Anwendungsfälle zur Verfügung zu stellen.

Der Nachteil von mehreren Views ist aber die kognitive Belastung für den Betrachter. Er muss den Zusammenhang der Darstellungen verstehen und nicht nur technisch, sondern auch gedanklich in der Lage sein, zwischen diesen zu wechseln [61].

Desweiteren sollte Visualisierung nicht nur das Artefakt (also die Software) betrachten, sondern auch die menschlichen Tätigkeiten und sozialen Prozessen bei der Softwareentwicklung berücksichtigen. Die Abbildung solcher Aspekte kann Informationen liefern, die zum Verständnis des Software-Designs beitragen [54].

Eine weitere Herausforderung bei der Konzeption von Softwarevisualisierungen ist es, herauszufinden, was genau visualisiert werden muss, um den Benutzer bei seiner Tätigkeit zu unterstützen. Diese Frage ist nicht immer leicht zu beantworten, weil das, was visualisiert werden *kann*, sich nicht unbedingt mit dem deckt, was idealerweise visualisiert werden *sollte* [62].

3.2.3 Klassifizierung

Grundsätzlich können Visualisierungen danach unterschieden werden, welchen dieser drei Aspekte der Software sie visualisieren [16]:

1. Struktur
2. Evolution
3. Verhalten

Die ersten beiden Aspekte bedürfen einer statischen Analyse, während für letzteren eine dynamische Analyse (vgl. Abschnitt 2.4) nötig ist. Da im Rahmen dieser Arbeit nur die statischen Eigenschaften von Software betrachtet werden, sollen im Folgenden auch nur Visualisierungen für strukturelle und evolutionäre Aspekte vorgestellt werden.

Caserta und Zendra [11] liefern einen umfassenden Überblick über den Stand der Technik im Bereich der Visualisierung von statischen Softwareaspekten. Sie klassifizieren die Visualisierungsmethoden zusätzlich nach ihrer Granularität basierend auf dem Abstraktionslevel, das sie darstellen:

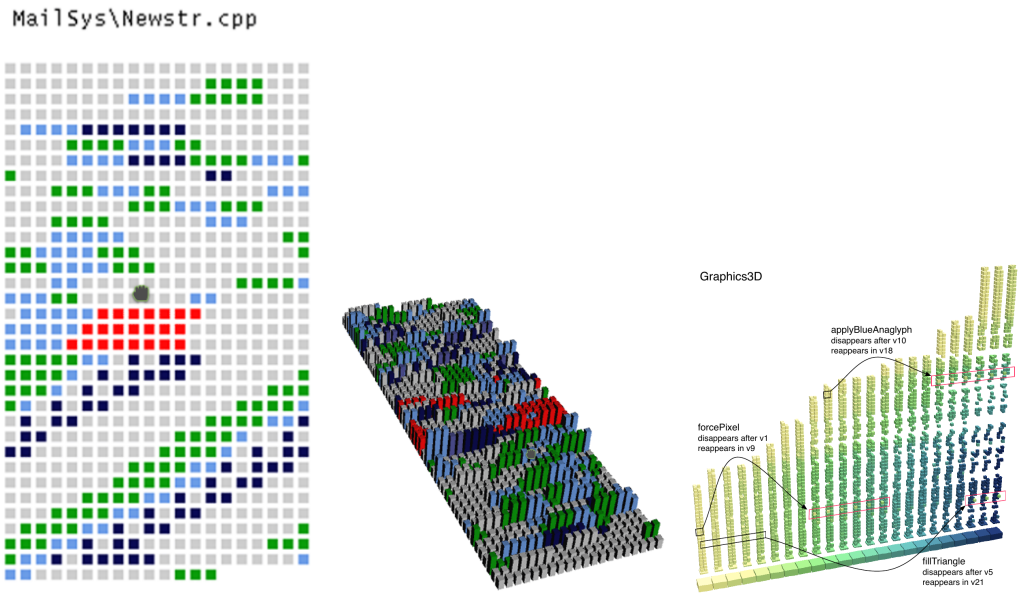
1. Source-Code-/Instruktionsebene
2. Package-, Klassen- und Methodenebene
3. Architekturebene

Neben der zweidimensionalen Softwarevisualisierung, konzentriert sich ein großer Teil der Forschung in den letzten Jahren auch auf 3D-Visualisierung. Dreidimensionale Visualisierung kann, zum Beispiel bei der Exploration großer Strukturen, Vorteile haben. Gleichzeitig bringt sie aber auch neue Herausforderungen mit sich, vor allem hinsichtlich der Navigation im dreidimensionalen Raum, die mehr Freiheitsgrade besitzt als im Zweidimensionalen [57].

3.2.4 Beispiele

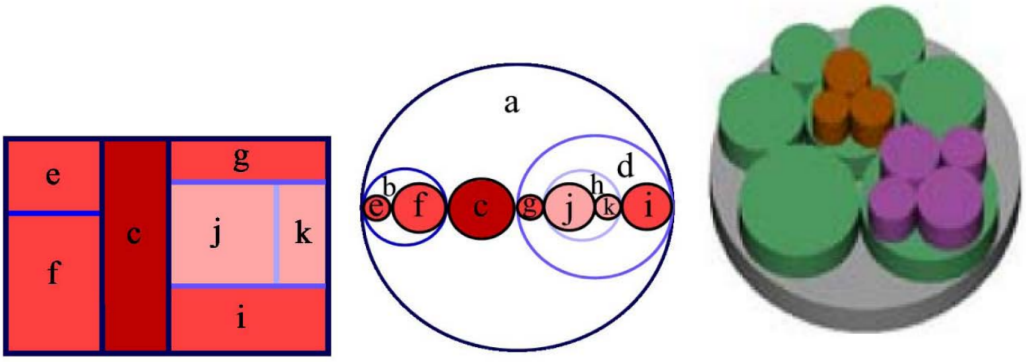
Es gibt unzählige Arten von Darstellungen, die für die Visualisierung von Software eingesetzt werden [11]. An dieser Stelle kann nur ein grober Überblick mit einigen Beispielen gegeben werden.

Besonders kompakt sind Pixeldarstellungen in 2D und 3D, die vor allem auf Instruktionsebene angewandt werden (Abbildungen 3.4a und 3.4b). Sie können neben der Zeilenlänge und der Einrückung auch sprachliche Konstrukte wie Verzweigungen und Schleifen (über die Farbe) und zusätzliche Metriken (über die dritte Dimension) anzeigen. Zur Visualisierung von evolutionären Aspekten, wie die Modifikationen von Klassen, gibt es Pixeldarstellungen, in denen die X-Achse als Zeitachse dient (Abbildung 3.4c). Gemein ist diesen Darstellungen, dass sie ohne ausführliche Legende schwer zu interpretieren sind, da sie keine allgemeingültigen Metaphern benutzen.



(a) (b) (c)

Abbildung 3.4: Pixelbasierte Darstellungen von Quellcode. (Quellen: [41, 81]).



(a) (b) (c)

Abbildung 3.5: Treemap-Darstellungen (Quelle: [11]).

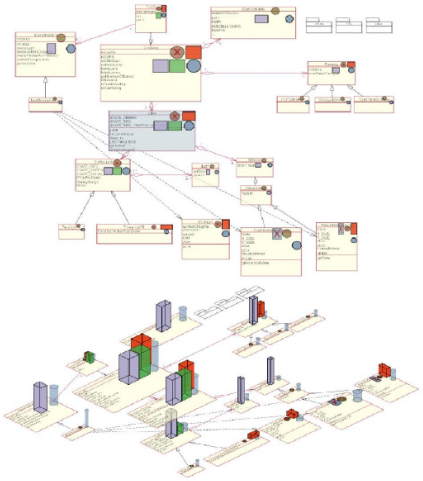


Abbildung 3.6: Klassendiagramme in Kombination mit anderen Diagrammen (Quelle: [75]).

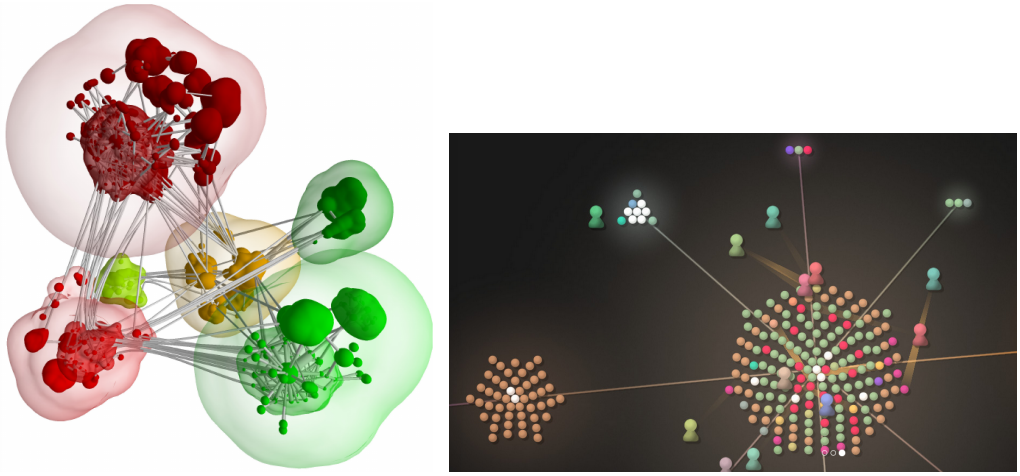
Zur Darstellung von Strukturen auf höherer Abstraktionsebene werden häufig verschiedenste Varianten der Treemap eingesetzt (Abbildung 3.5). Eine Treemap stellt eine Baumstruktur dar, wobei besonders die Größe der Knoten prägnant visualisiert wird. Im Vergleich zu einer Graph-Darstellung, ist die Treemap gut geeignet, wenn nur eine begrenzte Zeichenfläche zur Verfügung steht, da sie den vorhandenen Platz komplett ausnutzt [31]. Hierarchische Ordner-, Package- und Modulstrukturen sind für diese Darstellungsart geeignet.

Um Beziehungen in einer Architektur darzustellen, werden häufig Netze bzw. Graphen verwendet. Eine klassische und standardisierte Methode ist das UML Klassendiagramm, welches bei großer Anzahl an Klassen schnell an seine Grenzen stößt. Weiterentwicklungen von UML-Diagrammen legen zusätzlich andere Diagramme über die Klassen, um neben den Beziehungen, weitere Informationen vermitteln zu können (Abbildung 3.6).

Dem Problem der Skalierbarkeit wird zum Teil mit Graphen begegnet, die geclustert werden, um mit verschiedenen Levels of Detail die Informationsdichte bei Bedarf zu reduzieren. Auch dreidimensionale Graphen kommen dabei zum Einsatz (Abbildung 3.7a). Ein anderer Ansatz ist die Animation von Graphen, um die Entwicklungshistorie eines Projekts über die Zeit zu visualisieren (Abbildung 3.7b.

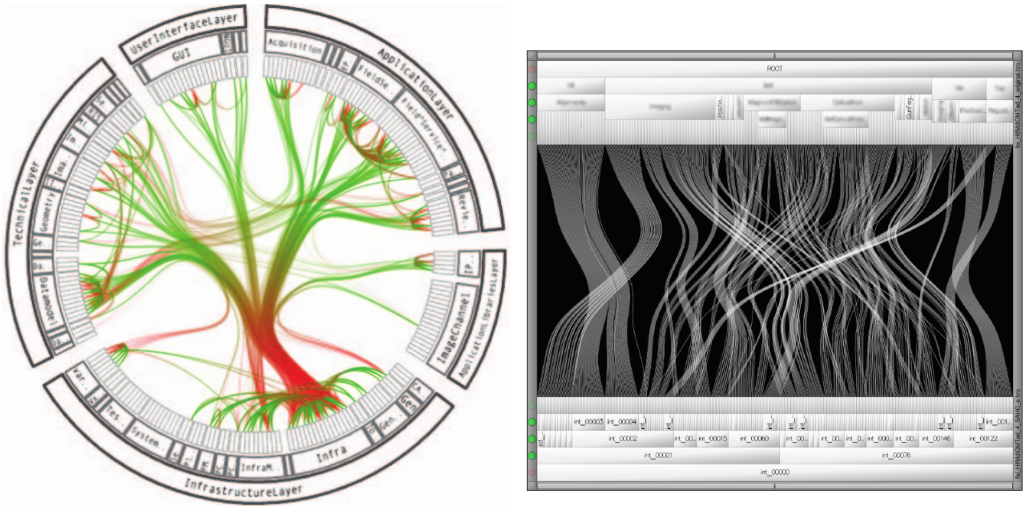
Neben den genannten Graphen, gibt es auch alternative Netzdarstellungen für Beziehungen zwischen Komponenten. Abbildung 3.8a zeigt eine kreisförmige Anordnung, die sogenanntes *Hierarchical Edge Bundling* benutzt, um die sehr große Zahl an Verbindungen zu verringern. Das, in Abbildung 3.8b gezeigte, Netz stellt hingegen zwei Versionen einer Hierarchie von Softwarekomponenten gegenüber, um diese vergleichen zu können.

Neben den zweidimensionalen Graphen und Netzen, kommen immer öfter sogenannte *Real-World-Metaphern* zum Einsatz (Abbildung 3.9). Sie benutzen an Stelle von abstrakten grafischen Elementen Objekte aus der realen Welt, wie zum Beispiel Städte mit Häusern und Straßen, Landkarten oder Planetensysteme und werden



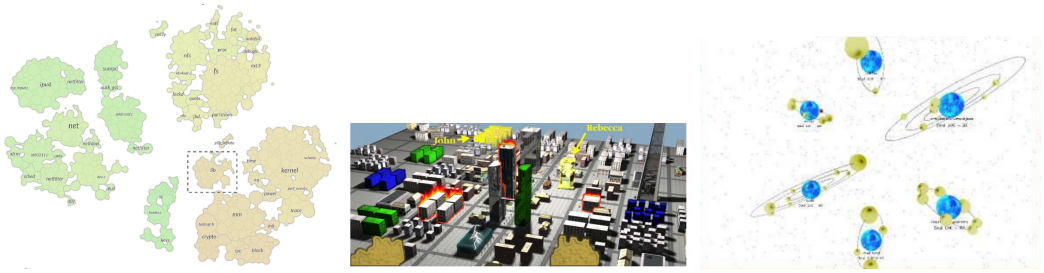
(a) (b)

Abbildung 3.7: Dreidimensionaler geclusterter Graph und animierte Versionshistorie als zweidimensionaler Graph (Quellen: [4, 93]).



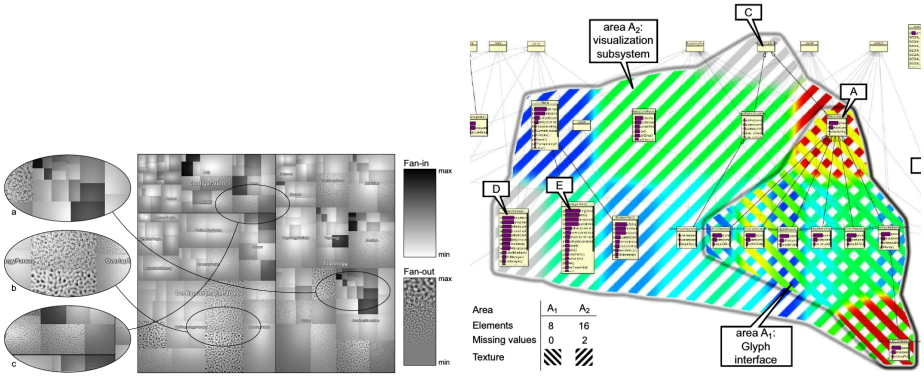
(a) (b)

Abbildung 3.8: Netzdarstellungen mit Hierarchical Edge Bundling (Quellen: [28, 29]).



(a) (b) (c)

Abbildung 3.9: Real-World-Metaphern: Ländern und Kontinente, Städte und Sonnensysteme (Quellen: [26, 55, 24]).



(a) (b)

Abbildung 3.10: Texturen und Farben zur Darstellung von Metriken (Quellen: [27, 10]).

meistens dreidimensional dargestellt. Durch die Vertrautheit mit den Objekten verspricht man sich von diesem Ansatz eine bessere Zugänglichkeit für den Betrachter und eine intuitive Interpretation der Informationen [68].

Unabhängig vom zugrunde liegenden Layout der Darstellungsart, lässt sich beobachten, dass häufig grafische Variablen, wie Farbe und Textur, benutzt werden, um Zusatzinformationen oder Metriken anzuzeigen (Abbildung 3.10).

3.2.5 Vorhandene Visualisierungsprogramme

Die im vorherigen Abschnitt vorgestellten Beispiele von Softwarevisualisierungen stammen größtenteils aus akademischen Anwendungen. Eine Vorstellung der einzelnen Tools würden den Rahmen dieser Arbeit sprengen. Eine tabellarische Übersicht über eine Auswahl von Anwendungen, sowohl kommerzielle, akademisch, als auch quelloffene, ist in [16] zu finden.

Bei der Recherche zu dieser Arbeit fiel vor allem auf, dass es nur äußerst wenige Software-Visualisierungstools gibt, die frei verfügbar sind und direkt auf ein eigenes Softwareprojekt angewandt werden könnten.

Verbreitete und gut zugängliche Tools, wie zum Beispiel Gource [93, 12], lie-

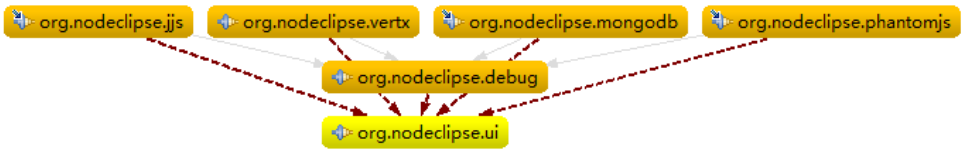


Abbildung 3.11: Visualisierung von OSGi-Bundles in einem Eclipse-Plugin (Quelle: [98]).

fern zwar beeindruckende visuelle Ergebnisse, aber wenig Mehrwert aus Sicht des Software-Engineering. Ein Erkenntnisgewinn ist damit nur begrenzt möglich.

Desweiteren gibt es bisher keine Visualisierungen, die in der Lage sind, OSGi basierte Java-Anwendungen umfassend darzustellen. Mit *PDE Incubator Dependency Visuzalization* [98] gibt es zwar ein Eclipse-Plugin, dass Abhängigkeiten zwischen OSGi-Bundles als Graph anzeigt (Abbildung 3.11), es bietet aber weder Interaktionsmöglichkeiten noch darüber hinausgehende Informationen, wie Metriken. Zudem beschränkt es sich auf die Abstraktionsebene von Bundles und wird offenbar nicht aktiv weiterentwickelt.

4 Konzept

Im Folgenden wird das Konzept erläutert, welches als Grundlage für die, im nächsten Kapitel beschriebene Realisierung, dient. Aufbauend auf den Zielen, die in der Einleitung genannt wurden, wird der Fokus der Arbeit genauer festgelegt. Desweiteren wird analysiert, aus welchen Quellen Informationen über eine Software gewonnen werden können. Anschließend wird darauf eingegangen, welche Aspekte von Softwarearchitekturen visualisiert werden sollen und warum diese Auswahl getroffen wurde. Es folgt eine Skizzierung des grundlegenden Aufbaus der Implementierung.

4.1 Zielgruppe

Wie in 2.4 und 2.5 gesehen, gibt es viele Aspekte von Software, die analysiert und letztendlich mit Hilfe diverser Metriken auch visualisiert werden können. Visualisierungen können ganz verschiedene Zwecke erfüllen. Damit sie von Nutzen sind, muss ihr Fokus möglichst klar definiert sein. Dafür müssen sowohl Zielgruppe, als auch Ziele festgelegt werden.

Die Zielgruppe von Softwarevisualisierung kann von fachfremden Personen, wie Managern, über neue Mitarbeiter bis hin zu Entwicklern der Software mit langjähriger Erfahrung reichen. Aufgrund der unterschiedlichen Kenntnisse und Erwartungen dieser Gruppen, ist es selten möglich, allen gleichermaßen gerecht zu werden. Deshalb sollte hier eine Eingrenzung stattfinden.

In dieser Arbeit sind Entwickler, die aktiv an der Entwicklung der Software beteiligt sind, die primäre Zielgruppe. Fachfremde Personen, bzw. Personen, die nicht direkt im Entwicklungsprozess involviert sind, werden nicht berücksichtigt.

Die Visualisierung soll die Software-Entwickler dabei unterstützen,

- einen Überblick über die Softwarearchitektur und den Entwicklungsprozess zu bekommen. Dieser ist immer wichtig, auch wenn in großen Projekten niemand mehr den kompletten Code im Detail kennen kann.
- bestehende Strukturen in der Architektur zu erkennen, zu verstehen und gegebenenfalls zu überdenken.
- Schwachstellen oder sog. Anti-Pattern im Design zu erkennen.
- generell ein besseres „Gefühl“ für den Aufbau der Software zu bekommen

Dies alles kann eine gute Informations-Grundlage für Entscheidungen im Entwicklungsprozess bieten. Insgesamt soll so ein Beitrag zur Qualitätssicherung geleistet werden.

Neben bestehenden Entwicklern werden neue Teammitglieder als sekundäre Zielgruppe berücksichtigt. Diese müssen sich mit der Software und deren Aufbau vertraut machen. Hier liegt der Fokus weniger auf dem Erkennen von Schwachstellen oder der Verbesserung der Software, als viel mehr auf dem Gewinnen eines Überblicks, dem Erkennen von Strukturen und dem allgemeinen Verständnis der Architektur.

Die Vermutung liegt nahe, dass sich große Teile der interessanten Aspekte bei beiden Zielgruppen überschneiden. In der Evaluierung (Kapitel 6) werden User-Stories verwendet, in denen sowohl bestehende Entwickler, als auch neue Entwickler berücksichtigt werden.

4.2 Fokus

Aufgrund der ausgewählten Zielgruppe ist es sinnvoll, sich auf interne Attribute der zu analysierenden Software zu konzentrieren (siehe Abschnitt 2.5). Hier ist das Attribut der Modularität besonders interessant, da komponenten- und servicebasierte Entwicklung, also Modularisierung, ein Hauptfeature des OSGi-Frameworks ist.

Modularität kann auf verschiedenen Ebenen betrachtet werden: von Klassen über Packages bis hin zu Bundles. Modularisierung bedeutet immer auch, dass Module in Beziehung zueinander stehen. Deshalb soll der Fokus dieser Arbeit auf Beziehungen und Abhängigkeiten zwischen Komponenten liegen. Die Auswahl der Darstellungsarten in der Visualisierung sollte sich an diesem Fokus orientieren. Trotzdem können andere Aspekte, wie Größe, Historie und Kohäsion, als Zusatzinformationen integriert werden.

Die Visualisierung soll dort eingesetzt werden, wo grafische Darstellung wirklich einen Mehrwert gegenüber anderen Darstellungen bietet. Metriken werden zwar mit einbezogen, aber der Fokus liegt nicht auf der Präsentation von Metriken, sondern darauf, eine interaktive Exploration der Software zu ermöglichen. Exploration von Daten hilft dabei, Muster zu erkennen und so deren Struktur zu verstehen [79].

Wie in Kapitel 3 beschrieben, können grafische Darstellungen besonders im Hinblick auf Skalierbarkeit von Vorteil sein. Deshalb liegt das Augenmerk der Visualisierung nicht darauf so viele Informationen wie möglich darzustellen (auch wenn die Daten zur Verfügung stehen). Stattdessen ist das Ziel, Informationen geschickt zusammenzufassen und dem Benutzer die Möglichkeit zu geben, auszuwählen, an welchen Stellen er mehr Details sehen möchte.

Wie im vorherigen Abschnitt erwähnt, soll die Softwarevisualisierung einen Beitrag zur Qualitätssicherung leisten. Trotzdem soll es nicht ihre Aufgabe sein, Schwachstellen oder Ähnliches automatisch zu erkennen und als solche zu präsentieren. Dem Benutzer soll durch visuelle Exploration der Software die Möglichkeit gegeben werden, diese Stellen selber zu finden, wenn das seiner Intention entspricht. Automatisierte Fehleranalyse hingegen soll nicht Thema dieser Arbeit sein.

Auch eine Bewertung des Code-Designs wäre denkbar. Die meisten in Abschnitt 2.6 beschriebenen Metriken sind isoliert betrachtet aber zu feingranular, um quantifizierte Aussagen über Design-Aspekte treffen zu können. Die Umsetzung von metrikbasierten Regeln zur Erkennung von Design-Fehlern, wie sie Marinescu [42] entwickelt

hat, wäre jedoch zu aufwändig.

Desweiteren könnte man zwar auf Anti-Pattern oder Verstöße gegen Guidelines aufmerksam machen, allerdings ist es schwer sich darauf festzulegen, was gutes bzw. schlechtes Design ist. Softwareprojekte sind sehr verschieden und es gibt wenig allgemeingültige Guidelines.

Häufig zitierte „Best Practices“ im OSGi-Umfeld sind beispielsweise [13]:

> *„Version that which is versionable.“*
>
> *„Separate API from implementation.“*
>
> *„Avoid split packages and Require-Bundle.“*

Diese Richtlinien erscheinen im Hinblick auf lose Kopplung von Komponenten grundsätzlich als sinnvoll. In vielen Projekten gibt es aber durchaus gute Gründe, die dafür sprechen, sich nicht danach zu richten.

Deshalb soll die Visualisierung keine Wertung der Softwarequalität vornehmen. Die Interpretation der dargestellten Daten soll durch den Benutzer geschehen. Dafür spricht auch, dass der Mensch generell besser Muster erkennen und Semantik mit visuellen Darstellungen verknüpfen kann, als Computer [84]. Voraussetzung dafür ist, dass die nötigen Informationen geeignet visuell aufbereitet werden.

4.3 Informationsquellen

Wie schon in Kapitel 1 erwähnt, ist der Java-Quellcode nur eine von vielen Quellen durch die man Informationen über eine Software gewinnen kann. Bevor festgelget werden kann, welche Aspekte der Softwarearchitektur visualisiert werden sollen, muss analysiert werden, welche Datenquellen zur Verfügung stehen. Zudem muss betrachtet werden, welche Schnittstellen diese zur Verfügung stellen und ob darüber eine automatische Extraktion der Daten möglich ist.

Die Datenquellen lassen sich allgemein in drei Arten einteilen:

1. Produkte der Softwareentwicklung, das heißt, Daten, die Teil der Software sind. Vor allem Quellcode und das kompilierte Programm, aber auch Spezifikationen, welche die Software beschreiben (vgl. „Produkt-Entitäten“ in Abschnitt 2.5).

2. Werkzeuge, die im Software-Entwicklungsprozess benutzt werden und Daten über den Prozess speichern. Dabei handelt es sich vor allem um Metadaten, die nicht Teil der Software selbst sind (vgl. „Prozess-Entitäten“ in Abschnitt 2.5).

3. Dateien oder Datenbanken mit Daten, die zur Laufzeit der Software, im Produktiv- oder Testbetrieb gesammelt werden.

Spezifikationen, wie sie im ersten Punkt genannt wurden, werden im Folgenden nicht weiter beachtet. Sie enthalten zwar wichtige Informationen über eine Software,

doch ihre automatische Analyse und Visualisierung ist äußerst schwierig. Während Quellcode von Natur aus gut maschinenlesbar ist, sind Spezifkationen und Design-Dokumente meistens ausschließlich für das Lesen durch Menschen gedacht. Außerdem ist es in der Praxis ein großes Problem, dass Spezifikationen nicht kontinuierlich an den aktuellen Stand der Entwicklung angepasst werden und somit Spezifikation und Implementierung nicht mehr konsistent zueinander sind [8].

Im Folgenden werden einige typische Datenquellen beschrieben, die in OSGi-Projekten zu Verfügung stehen können:

Java Quellcode Quellcode ist die offensichtlichste und reichaltigeste Quelle an Informationen über eine Software. Zwar ist Java-Code darauf ausgelegt, von Menschen geschrieben und gelesen zu werden, da er aber von einem Compiler vor der Ausführung in Bytecode übersetzt werden muss, besitzt er natürlich eine Grammatik, die gut von Parsern verstanden werden kann. Deshalb gibt es neben dem Java-Compiler eine Vielzahl von Anwendungen und Bibliotheken, die Java-Code nicht nur parsen, sondern auch für einen bestimmten Zweck analysieren. Die Informationsgewinnung sollte an dieser Stelle also kein Problem darstellen.

Java Bytecode Bytecode ist eine kompilierte Form von Java-Code. Da er von der JVM interpretiert wird, enthält er aber einen kleinen und leicht zu verstehenden Befehlssatz [15]. Daher kann er vergleichsweise einfach analysiert werden und bietet ähnliche Möglichkeiten wie die Analyse von Quellcode. Zu Analyse von Bytecode stehen diverse Bibliotheken zur Auswahl [9, 35].

Manifest-Dateien Im Fall von OSGi-Anwendungen enthalten die Manifest-Dateien wichtige Informationen über die Abhängigkeiten und Versionen von Komponenten. Das einfache textbasierte Dateiformat aus Schlüssel-Wert-Paaren bietet sich besonders gut für eine Auswertung an.

Service-Deklarationen Neben den Manifest-Dateien sind die Service-Deklarationen die zweite Art von Konfigurationsdateien bei OSGi, die wesentliche Informationen zur Softwarearchitektur enthalten. Das gängie XML-Dateiformat macht eine Analyse ebenfalls einfach.

Versionskontrollsystem In nahezu jedem Softwareprojekt werden Systeme wie *Subversion* oder *git* zur Versionskontrolle eingesetzt. Diese führen Buch über alle Änderungen an Dateien, inklusive der Autoren und bieten sich daher als Quelle für Informationen über die Historie der Software an. Versionskontrollsysteme speichern ihre Daten meist intern in komprimierter Form und (bei zentralen Systemen) ausschließlich serverseitig. Vorgesehen ist meist nur eine interaktive, kommandozeilenbasierte, Schnittstelle. Es gibt aber Bibliotheken, die eine API zur programmatischen Arbeit mit Repositories zu Verfügung stellen.

Bug-Tracker Bug-Tracking-Systeme, die üblicherweise als Webanwendung laufen, enthalten interessante Metadaten zu einer Software. Aus diesen wird beispielsweise deutlich, an welchen Stellen im Code häufig Fehler auftreten und wer an

der Behebung der Fehler mitwirkt. Oft ist eine Verknüpfung zum Versionskontrollsystem vorhanden, so dass diese Daten kombiniert werden können. Viele Bug-Tracking-Systeme besitzen eine API. Der verbreitete *Mantis Bug Tracker* hat zum Beispiel eine SOAP-Schnittstelle, über die Daten abgefragt werden können [101].

Build-Management-System In vielen Java-Projekten kommen Werkzeuge zur Build-Automatisierung zum Einsatz, welche beim Kompilieren komplexer Software helfen. Sie lösen externe Abhängigkeiten auf und fügen alle Bestandteile der Software (Code, Konfigurationsdateien, sonstige Ressourcen, etc.) zu einem ausführbaren Programm zusammen. Im Java-Umfeld sind unter anderem Ant, Maven und Gradle verbreitete Build-Management-Systeme. Sie besitzen viele Informationen über die Struktur der Software, die aus ihren textbasierten Konfigurationsdateien gewonnen werden könnten.

Continuous Integration Server Systeme wie *Jenkins*, die serverseitig fortlaufende Kompilierung und automatisierte Tests ausführen, können möglicherweise relevante Metadaten zum Software-Entwicklungsprozess liefern. Jenkins beispielsweise stellt dazu eine REST-Schnittstelle zur Verfügung [99].

Testergebnisse Software-Tests spielen eine große Rolle bei der Qualitätssicherung und können in der Regel sogenannte *Reports* generieren, die Testergebnisse enthalten. Diese könnten bei der Softwareanalyse mit einbezogen werden, um Informationen über die Qualität von Programmkomponenten und die Testabdeckung zu erhalten. Ein weit verbreitetes XML-basiertes, maschinenlesbares Format für Test-Reports ist *JUnit XML*.

Profiling Sogenannte *Profiler* führen eine dynamische Analyse von Programmen durch und können eine Vielzahl an Informationen, wie Speicherverbrauch und Call-Graphen, gewinnen. Besonders ein Call-Graph kann interessante Erkenntnisse bei Betrachtung einer Architektur bringen. Inwiefern Profile-Ergebnisse programmatisch ausgewertet werden können, ist aber stark vom eingesetzten Profiler abhängig.

Log-Dateien Viele Programme schreiben während der Ausführung Informationen, die zum Debuggen genutzt werden können in Log-Dateien. Sie enthalten semantische Informationen über das Laufzeitverhalten.

Personen Softwareentwickler und andere Personen lassen sich in keine der drei oben genannten Arten von Datenquellen einordnen. Trotzdem können sie Informationen zur Verfügung stellen, die nirgends in einer anderen Form vorliegen und nur schwer bis gar nicht automatisiert ermittelt werden können. Dies sind vor allem semantische Informationen, wie zum Beispiel die Zuordnung von Code zu Funktionalität (GUI, Utilities, Datenhaltung . . .). Denkbar wäre außerdem, dass historische Daten um Informationen zu Entwicklungsphasen des Projekts (Konzeption, Entwicklung, Test, Wartung . . .) ergänzt werden. Der Nachteil dieser Art der Informationsbeschaffung liegt auf der Hand: Sie findet manuell statt und lässt sich schwer verallgemeinern.

Im Rahmen dieser Arbeit können nicht alle möglichen Informationsquellen berücksichtigt werden. Stattdessen muss eine Auswahl getroffen werden, die in der Summe ausreichend Daten für eine sinnvolle Visualisierung liefern.

Die Auswahl wurde wie folgt getroffen: Das Hauptaugenmerk liegt auf den statischen Daten. Neben dem Java-Quelltext werden auch Manifest-Dateien und Service-Deklarationen einbezogen, da diese in OSGi-Anwendungen wichtige Informationen über die Architektur beinhalten. Programmcode in seiner kompilierten Form (Bytecode) wird nicht analysiert, da er keinen Mehrwert an Informationen gegenüber dem Quelltext bietet. Diese Entscheidung hängt auch mit den verfügbaren Analyse-Bibliotheken zusammen, auf die im nächsten Kapitel näher eingegangen wird.

Zusätzlich zu diesen Datenquellen, welche die Software selbst ausmachen, sollen aus dem Versionskontrollsystem Informationen über die Historie des Quelltextes und Metadaten über Autoren und den Entwicklungsprozess gewonnen werden. In dieser Hinsicht würde das zugehörige Bug-Tracking System zwar weitere Informationen liefern, doch dies würde den Rahmen dieser Arbeit sprengen.

Auch andere externe Werkzeuge, wie Build- und Continuous-Integration-Systeme, werden von der Analyse ausgeschlossen. Ziel ist es aber, die Implementierung so erweiterbar zu halten, dass diese Informationsquellen später integriert werden können.

Da der Fokus der Arbeit auf der Architektur von Software liegt und diese in der Regel statisch ist, werden auch keinerlei dynamische Daten, für die eine Ausführung des Programms nötig ist, berücksichtigt.

4.4 Aspekte

Im Folgenden werden die Aspekte beschrieben, die mit Blick auf den Fokus der Arbeit von Interesse sind und für die die ausgewählten Informationsquellen ausreichend Daten zur Verfügung stellen. Eine Übersicht ist in Tabelle 4.1 zu sehen. Darin ist auch angegeben, inwieweit die einzelnen Aspekte im praktichen Teil dieser Arbeit (siehe Kapitel 5) berücksichtigt wurden.

Abhängigkeiten

Komponenten und deren strukturelle Abhängigkeiten sollen auf veschiedenen Abstraktionsebenen visualisiert werden: von Klassen über Packages bis hin zu Bundles. Dabei kann nach Art der Abhängigkeit, wie in Abschnitt 2.3.1 beschrieben, unterschieden werden. Wo es sinnvoll erscheint, könnten auch transitive Abhängigkeiten dargestellt werden.

Modularität

Die Stärke der Abhängigkeiten, das heißt, die Kopplung zwischen den dargestellten Komponenten, soll visualisiert werden. Metriken dafür wurden in Abschnitt 2.6.2 vorgestellt.

Bei Bundles ist es interessant zu sehen, wie viel Code öffentlich, also exportiert, ist und wie viel Code intern gekapselt wird. Durch die `Export-Package`-Deklaration

	Aspekt	Implementierung	Anmerkungen
Abhängigkeit	Abhäng. zwischen Klassen	✓	
	Abhäng. zwischen Packages	✓	
	Abhäng. zwischen Bundles	✓	
	Art der Abhängigkeit	✓	`Import`, `RequireBundle`
	transitive Abhängigkeiten	◯	
Modularität	Kopplung	✓	Metriken: siehe 2.6.2
	Kohäsion	◯	Metriken: siehe 2.6.3
	interner vs. öffentlicher Code	✓	`Export-Package`
	importierter vs. benutzter Code	◯	
	Exports vs. Imports	✓	
	Strukturen im Modulgraph	◯	Cluster, Ebenen ...
Größe	Größe von Bundles	✓	Metriken: siehe 2.6.1
	Größe von Packages	✓	Metriken: siehe 2.6.1
	Größe von Klassen	✓	Metriken: siehe 2.6.1
Services	Service-Übersicht	✓	`provide`
	Service-Nutzung	✓	`reference`
	Service-Implementierung	✓	`implements`
Historie	Evolutionäe Abhängigkeiten	◯	
	Hotspots	◯	
	Code-Owner	◯	
	Einordnung in Entwicklungsphasen	✕	
Semantik	Semantische Kopplung	✕	
	Anti-Pattern	✕	
	„Interessante“ Bundles	✕	

Tabelle 4.1: Aspekte einer Softwarearchitektur, die auf Basis der in Abschnitt 4.3 ausgewählten Informationsquellen, analysiert und visualisiert werden können. Zu jedem Aspekt wird angegeben, inwieweit er im Rahmen dieser Arbeit implementiert wurde (✓: visualisiert, ◯: nötige Daten extrahiert aber nicht visualisiert, ✕: nicht berücksichtigt).

im Bundle-Manifest kann dies auf der Granularität von Packages leicht ermittelt werden.

Genau wie das Exportieren, findet das Importieren bei Bundles ebenfalls auf Package-Ebene statt. Dadurch werden womöglich mehr Klassen importiert, als tatsächlich genutzt. Dieses Verhältnis kann gleichermaßen aufschlussreiche Informationen über die Modulstruktur geben.

Betrachtet man Komponenten und deren Beziehungen als Graph, können Strukturen in diesem Graph, zum Beispiel Zyklen, Baumstrukturen[1] und Cluster von Knoten, analysiert werden.

Services

Da Services einen zentralen Bestandteil von OSGi-Anwendungen und deren Modularität ausmachen, sollen diese natürlich auch einbezogen werden. Dabei müssen drei verschiedene Arten von Beziehungen zwischen Bundles und Services unterschieden werden: Anbieten, Implementieren und Referenzierung.

Größe

Neben den Beziehungen zwischen Komponenten, soll vor allem auch deren Größe unter Einsatz der in Abschnitt 2.6.1 genannten Metriken betrachtet werden. Hier gibt es wieder die Granularität von Klassen, Bundles und Packages.

Historie

Auch evolutionäre Aspekte können mit einbezogen werden. Neben der evolutionären Kopplung (Abschnitt 2.3.1) könnte auch dargestellt werden, wo im Code, über die Zeit gesehen, besonders viele Änderungen stattgefunden haben („Hotspots“).

Ein interessanter Aspekt wäre zudem die Zuordnung von Commits zu Entwicklungsphasen (Entwicklung, Testing, Pre-Release, Maintenance, o.Ä.). Die Idee dahinter ist, dass dies wertvolle Informationen für die Planung des Entwicklungsprozesses liefern könnte. Wird eine Komponente beispielsweise besonders in der Maintenance-Phase bearbeitet, deutet dies darauf hin, dass sie viele Fehler enthält und möglicherweise mehr Tests erstellt oder die Architektur generell überarbeitet werden sollte.

Semantik

Semantische Aspekte, wie die semantische Kopplung (Abschnitt 2.3.1) oder das Erkennen von Anti-Pattern, werden im Folgenden ausgeklammert, da sie entweder eine zu aufwändige Analyse oder das manuelle Bereitstellen von semantischen Informationen durch den Benutzer erfordern würden.

[1] siehe auch *Tree Impurity Measurement* [18]

4.5 Visualisierungsarten

Basierend auf den Aspekten, die betrachtet werden sollen, wurden mehrere geeignete Darstellungsarten für die Visualisierung ausgewählt. Durch diese soll jeweils einer der folgenden Hauptaspekte abgedeckt werden:

1. Abhängigkeiten zwischen Bundles
2. Beziehungen zwischen Service-Components
3. Package- und Klassenstruktur innerhalb von Bundles
4. Abhängigkeiten zwischen Klassen
5. Evolution von Komponenten

Somit kann die Software auf verschiedenen Abstraktionsebenen (auch *Levels-of-Detail*), von Bundles über Services bis hin zu Packages und Klassen, betrachtet werden. Weitere Aspekte und Metriken, wie zum Beispiel die Größe von Bundles oder die Stärke von Abhängigkeiten, sollen durch Anpassung der grafischen Variablen in die Darstellungen einbezogen werden.

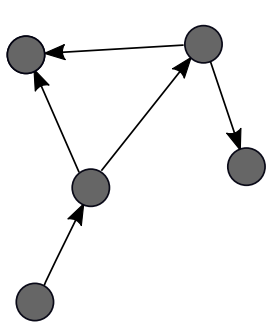
Abhängigkeiten zwischen Bundles

Bei den Abhängigkeiten zwischen Bundles geht es darum, Beziehungen zwischen Elementen der gleichen Informationskomponente darzustellen, weshalb sich nach Bertin dafür eine Netzdarstellung (vgl. Abschnitt 3.1.2) anbietet. Ein gerichteter Graph mit Bundles als Knoten und Abhängigkeiten als gerichtete Kanten, ist hier gut geeignet (Abbildung 4.1a). Ein Graph bietet eine gute Übersicht über Komponenten und ihre Abhängigkeiten und besitzt diverse grafische Variablen, die dazu genutzt werden können, zusätzliche Informationen anzuzeigen:

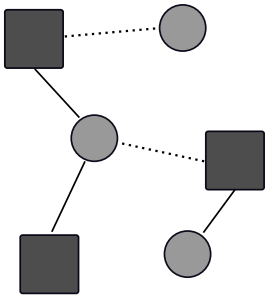
- Größe der Knoten
- Form der Knoten
- Farbe der Knoten
- Textur der Knoten
- Dicke der Kanten
- Form der Kanten
- Farbe der Kanten

Beziehungen zwischen Service-Components und Services

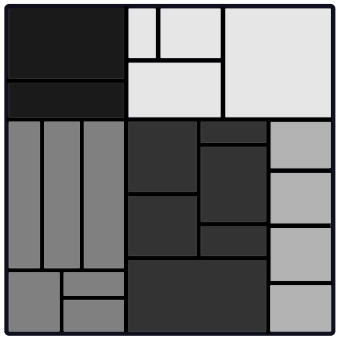
Bei der Betrachtung des Service-Aspekts gibt es zwei Arten von Komponenten: Service-Components und Services. Deren Beziehungen können ebenfalls als Graph dargestellt werden, wobei Service-Components und Services durch die Form der Knoten unterschieden werden können (Abbildung 4.1b). Auch von Beziehungen gibt es zwei Varianten, nämlich das Anbieten (*provide*) und die Benutzung (*reference*)



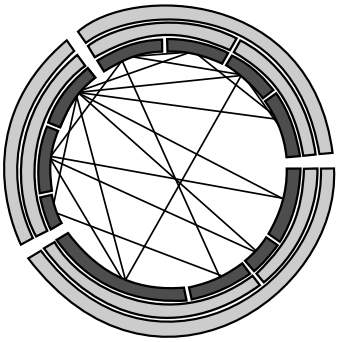
(a) Bundles



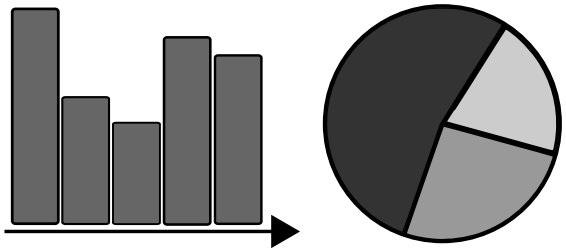
(b) Services und Service-Components



(c) Klassen und Packages (Struktur)



(d) Klassen und Packages (Abhängigkeiten)



(e) Evolution und Autoren

Abbildung 4.1: Ausgewählte Darstellungsarten für verschiedene Aspekte der Software.

eines Services durch ein Service-Component. Sie können durch die Form der Kanten (durchgezogen, gestrichelt) unterschieden werden. Da wahrscheinlich Service-Components aus mehreren Bundles gleichzeitig dargestellt werden, sollte es außerdem eine Zuordnung zu den Bundles geben, die zum Beispiel über die Farbe von Knoten umgesetzt werden kann.

Abhängigkeiten zwischen Klassen

Eine weitere Darstellung soll die Abhängigkeiten auf Klassenebene eines oder mehrerer Bundles visualisieren. Statt eines klassischen Graphen, wird dazu ein Netz verwendet, dass viereckige Knoten auf einem Kreis anordnet (Abbildung 4.1d). Der Vorteil davon ist, dass zusätzlich die Package-Hierarchie, als äußere Ringe, dargestellt werden kann. Außerdem kann *Hierarchical Edge Bundling* angewandt werden [28] (siehe Abbildung 3.8a). Sie bündelt Kanten zum Beispiel je Package und sorgt so für mehr Übersichtlichkeit bei großer Kantenanzahl.

Package- und Klassenstruktur innerhalb von Bundles

Bei der Betrachtung eines Bundles auf Package- und Klassenebene sind neben den Abhängigkeiten, vor allem die hierarchischen Strukturen und Größenverhältnisse von Interesse. Dafür sind Treemap-Darstellungen besonders gut geeignet (Abbildung 4.1c). Während die Größe der Blätter die Größe der zugehörigen Klassen darstellen, kann über die Farbe die Zuordnung zu Packages sichtbar gemacht werden. Zudem lässt sich das schon erwähnte Hierarchical Edge Bundling auch auf Treemaps anwenden, um Abhängigkeiten zwischen den Komponenten einzublenden [28].

Evolution von Komponenten

Informationen über die Evolution von Komponenten eignen sich zur Integration als Metrik in andere Darstellungen. Beispielsweise können Alter oder Änderungshäufigkeit gut durch Farben kodiert werden. Desweiteren können klassische Diagramme (Abbildung 4.1e) eingesetzt werden, um Daten, wie die Anzahl an Änderungen, über die Zeit zu betrachten (Balken-/Liniendiagramm) oder die Beteiligung von Autoren zu visualisieren (Kreisdiagramm).

4.6 Implementierungsansatz

Die Idee ist es, die Implementierung in zwei überwiegend unabhängige Anwendungen, die nacheinander ablaufen, aufzuteilen. Zuerst werden die Daten eines Projektes extrahiert, analysiert und in ein Modell transformiert[2] (Abbildung 1.1). Dieses Datenmodell kann anschließend visualisiert werden. In Abbildung 4.2 ist der Implementierungsansatz zu sehen, dessen Idee im Folgenden näher erläutert wird.

[2]Der Einfachheit halber werden diese Schritte im Folgenden unter dem Begriff „Analyse“ zusammengefasst.

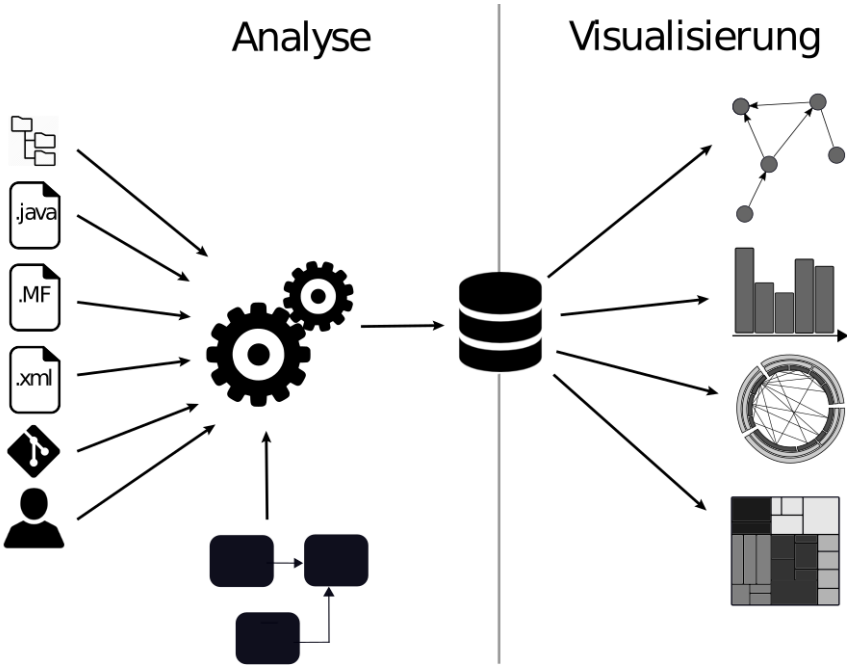


Abbildung 4.2: Implementierungsansatz: Die Analyse extrahiert Daten und erstellt ein Modell der Software. Die Visualisierung generiert verschiedene grafische Darstellungen davon.[3]

4.6.1 Aufteilung

Die Aufteilung der Implementierung in zwei Anwendungen hat den Vorteil, dass sie in unterschiedlichen Programmiersprachen und Laufzeitumgebungen geschrieben werden können. Da Analyse und Visualisierung sehr unterschiedliche technische Anforderungen stellen kann das hilfreich sein. Die Ausführung könnte nicht nur zeitlich, sondern auch räumlich getrennt auf verschiedenen Rechnern ablaufen. Beispielsweise wäre eine Ausführung der Analyse auf einem Server und die der Visualisierung auf einem Client denkbar.

Aufgabe des Analyse-Teils ist es, die verschiedenen Datenquellen eines Projekts auszulesen und die gewonnenen Daten in ein einziges Modell zu transformieren. Dieses Modell muss vorab durch ein Metamodell definiert werden. Anschließend wird das Modell der Visualisierung zur Verfügung gestellt.

Die Visualisierung ist dafür zuständig, dieses Modell auf verschiedene Arten grafisch darzustellen. Das Augenmerk an dieser Stelle liegt auf Erweiterbarkeit. Es soll möglich sein, jederzeit weitere Darstellungsarten hinzuzufügen. Die Darstellungen können unabhängig voneinander oder aber untereinander verknüpft sein.

4.6.2 Schnittstellen

Die einzige Schnittstelle zwischen Analyse und Visualisierung ist das Datenmodell. Eine direkte Kommunikation zwischen beiden Anwendungsteilen findet nicht statt,

[3]Bildquelle des Personen- und Datenbank-Icons: [92]

was zu einer sehr losen Kopplung führt. Es besteht also keine direkte Abhängigkeit von der Visualisierung zur Analyse, sondern nur von beiden Anwendungen zum Modell (Abbildung 4.3c). Die Analyse muss somit keine Programmierschnittstelle zur Verfügung stellen, was den Implementierungs- und Wartungsaufwand verringert. Dafür ist das Metamodell eine kritische Komponente. Ändert sich das Metamodell, müssen beide Anwendungsteile angepasst werden.

Vorbedingung für die Visualisierung ist, dass die Analyse wenigstens einmal ausgeführt wurde, damit die benötigten Daten (das Modell) vorliegen. Dieser entkoppelte Ablauf ist sinnvoll, da die Datenanalyse möglicherweise eine relativ lange Laufzeit hat, während die Visualisierung aber interaktiv und somit echtzeitfähig sein muss. Eine Online-Analyse ist also nicht praktikabel.

Zum Austausch des Modells muss dieses in einem Format vorliegen, dass die Analyse schreiben und die Visualisierung lesen kann. Desweiteren muss eine geeignete persistente Serialisierungsart ausgewählt werden. Einfache Text- oder Binärdateien sowie Datenbanken sind hier naheliegende Möglichkeiten.

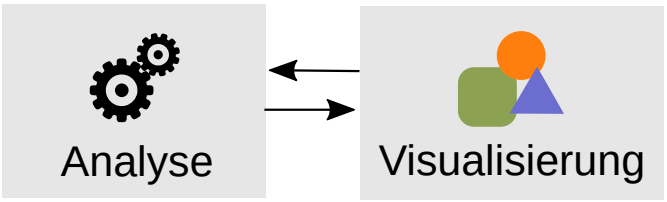
4.6.3 Alternativen

Eine Alternative zum beschriebenen Implementierungsansatz wäre es, alle Funktionen in einer Anwendung zu vereinen und die Teile nicht unabhängig voneinander auszuführen (Abbildung 4.3a). Ein Vorteil wäre, dass kein Datenaustausch und somit keine Serialisierung des Modells nötig wäre. Auch die Frage nach einem Datenformat fiele weg, da die komplette Verarbeitung „In-Memory“ stattfinden könnte.

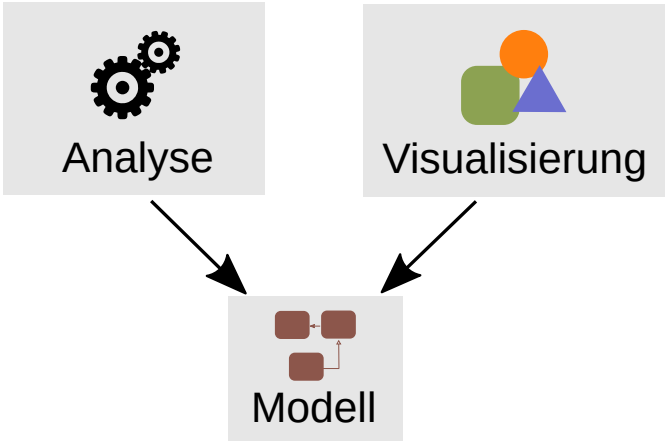
Andererseits könnte dies auch zu Speicherknappheit bei großen Datenmengen führen. Ein weiterer Nachteil dieses monolithischen Ansatzes wäre die Beschränkung auf eine einzige Programmiersprache und Laufzeitumgebung, wodurch man unter Umständen nicht allen technischen Anforderungen gerecht werden kann. Nicht zuletzt wäre dieser Ansatz weniger flexibel im Hinblick auf die Austauschbarkeit von Analyse und Visualisierung. Auch die Erweiterung der Darstellungsarten wäre aufwändiger, da eine geeignete Plugin-Struktur realisiert werden müsste.

Denkbar wäre auch ein Mittelweg aus den beiden oben beschriebenen Ansätzen: Analyse und Visualisierung könnten als voneinander getrennte Anwendungen laufen, wobei die Analyse als Server fungiert und die Visualisierung von diesem Daten abfragt (Abbildung 4.3b). Dagegen spricht allerdings ein vergleichsweise aufwändiges Setup der Anwendungen für den Nutzer und erhöhter Aufwand für die Implementierung eines Kommunikationsprotokolls (zum Beispiel über TCP).

(a) Monolithische Architektur



(b) Direkte Kommunikation zwischen Analyse und Visualisierung.



(c) Analyse und Visualisierung entkoppelt.

Abbildung 4.3: Mögliche Architekturen der Implementierung.

5 Umsetzung

In diesem Kapitel wird die Implementierung des oben beschriebenen Konzeptes erläutert. Sowohl konzeptionell, als auch technisch gesehen, lässt sich die Implementierung in drei Teile aufteilen: Metamodellierung, Analyse und Visualisierung. Diese werden im Folgenden separat behandelt. Es werden jeweils Ziel und Zweck erläutert sowie der Aufbau der Implementierung vorgestellt. Die Auswahl der Bibliotheken und andere Entscheidungen werden begründet. An ausgewählten Stellen wird auf Implementierungsdetails und offene Probleme eingegangen. Vor allem bei der Visualisierung konnten aus Zeitgründen nicht alle Ideen realisiert werden. Dort wird das Konzept aber soweit ausgearbeitet, dass auf dessen Grundlage in Zukunft eine Erweiterung der Implementierung möglich ist.

5.1 Metamodell

Wie schon in Abbildung 4.3 gezeigt, ist das Modell die gemeinsame Grundlage für Analyse und Visualisierung und stellt gleichzeitig die einzige Schnittstelle zwischen beiden Anwendungsteilen dar. Es handelt sich dabei um ein Modell der zu visualisierenden Sofware, das alle nötigen Daten für die Visualisierung beinhaltet. Um einen zuverlässigen Datenaustausch zu gewährleisten muss die Modellstruktur genau spezifiziert werden. Dies geschieht durch ein Metamodell .

5.1.1 Modellierungssprache und -werkzeug

Zu Beginn dieser Arbeit stellte sich die Frage, wie das Metamodell erstellt werden soll. Eine Möglichkeit wäre es, direkt Code (zum Beispiel Java) zu schreiben, der das Metamodell beschreibt. Dieser könnte in der Analyse direkt genutzt werden, um Modellinstanzen zu erzeugen. Die Modellierung direkt im Code ist aber nicht optimal, da sie fehleranfällig ist, bei großen Modellen unübersichtlich wird und keine grafische Darstellung bietet.

Diesen Nachteilen kann durch Nutzung einer (grafischen) Modellierungssprache, wie zum Beispiel UML, begegnet werden. UML-Editoren bieten eine kompakte grafische Darstellung, interaktive Modellierung und Generierung von Code (zum Beispiel POJOs[1]).

Die Wahl fiel jedoch auf das *Eclipse Modeling Framework* (EMF), dessen Funktionen deutlich über die eines einfachen UML-Editors hinausgehen.

EMF ist ein Framework für die Entwicklung von Anwendungen, die auf einem strukturierten Datenmodell aufbauen. Mit EMF erstellte Modelle heißen *Ecore-*

[1] Plain Old Java Objects

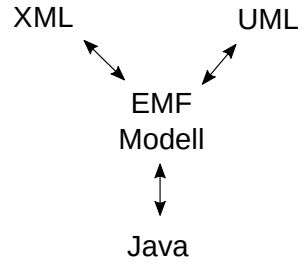


Abbildung 5.1: Ecore-Modell lassen sich in XML, UML oder Java-Code beschreiben und können zwischen diesen Formaten automatisch konvertiert werden (nach [71]).

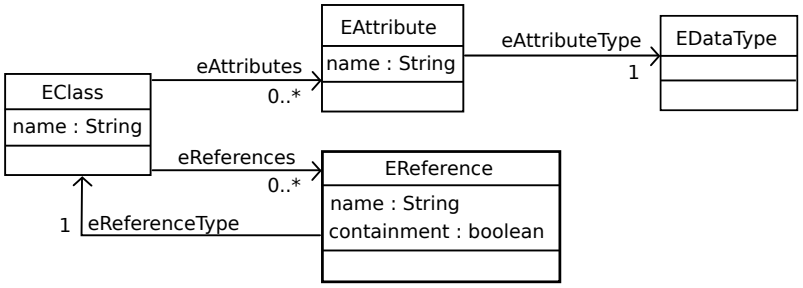


Abbildung 5.2: Ausschnitt des Ecore-Modells (nach [71]).

Modelle und werden in der Ecore-Metamodellierungssprache beschrieben. Ecore-Modelle können, müssen aber nicht, von Hand in dieser Sprache geschrieben werden. Sie können auch aus einer XML-Notation oder annotiertem UML generiert werden (Abbildung 5.1). Dazu bietet das Framework einen grafischen UML-Editor an, der in die Eclipse-IDE integriert ist und auf Knopfdruck Modelle erstellt. Aus dem Modell wiederum kann Java-Code generiert werden, der direkt in Java-Anwendungen eingebunden werden kann, um mit dem Modell zu arbeiten. Das manuelle Schreiben von Code ist an dieser Stelle also nicht nötig. Der Modellcode ist mehr als nur eine Menge von POJOs, da er unter anderem ein *Notification*-System beinhaltet, mit dem Änderungen an Modellinstanzen zur Laufzeit behandelt werden können. Diese Funktionalität wird im Folgenden aber nicht benötigt.

Dass Ecore-Modelle durch UML spezifiziert werden können, liegt daran, dass das Ecore-Metamodell eine Untermenge von UML ist [71]. Sie beschränkt sich auf das Modellieren von Klassen mit typisierten Attributen und Beziehungen (siehe Abbildung 5.2).

Ein weiterer Vorteil von EMF ist, dass es die Möglichkeit bietet, Modellinstanzen ohne zusätzlichen Aufwand zu serialisieren und zu speichern. Dieses Feature war ausschlaggebend dafür, dass bei der Implementierung dieser Arbeit die Wahl auf EMF fiel.

Die genannten Vorteile rechtfertigen die zusätzliche Komplexität, die durch die Verwendung von EMF entsteht. Ein Nachteil ist sicherlich, dass zur Änderung des Metamodells eine Eclipse-IDE mit installiertem Modelling-Framework nötig ist. Ist das Ecore-Modell aber einmal generiert, kann es in beliebigen Java-Anwendungen eingebunden werden.

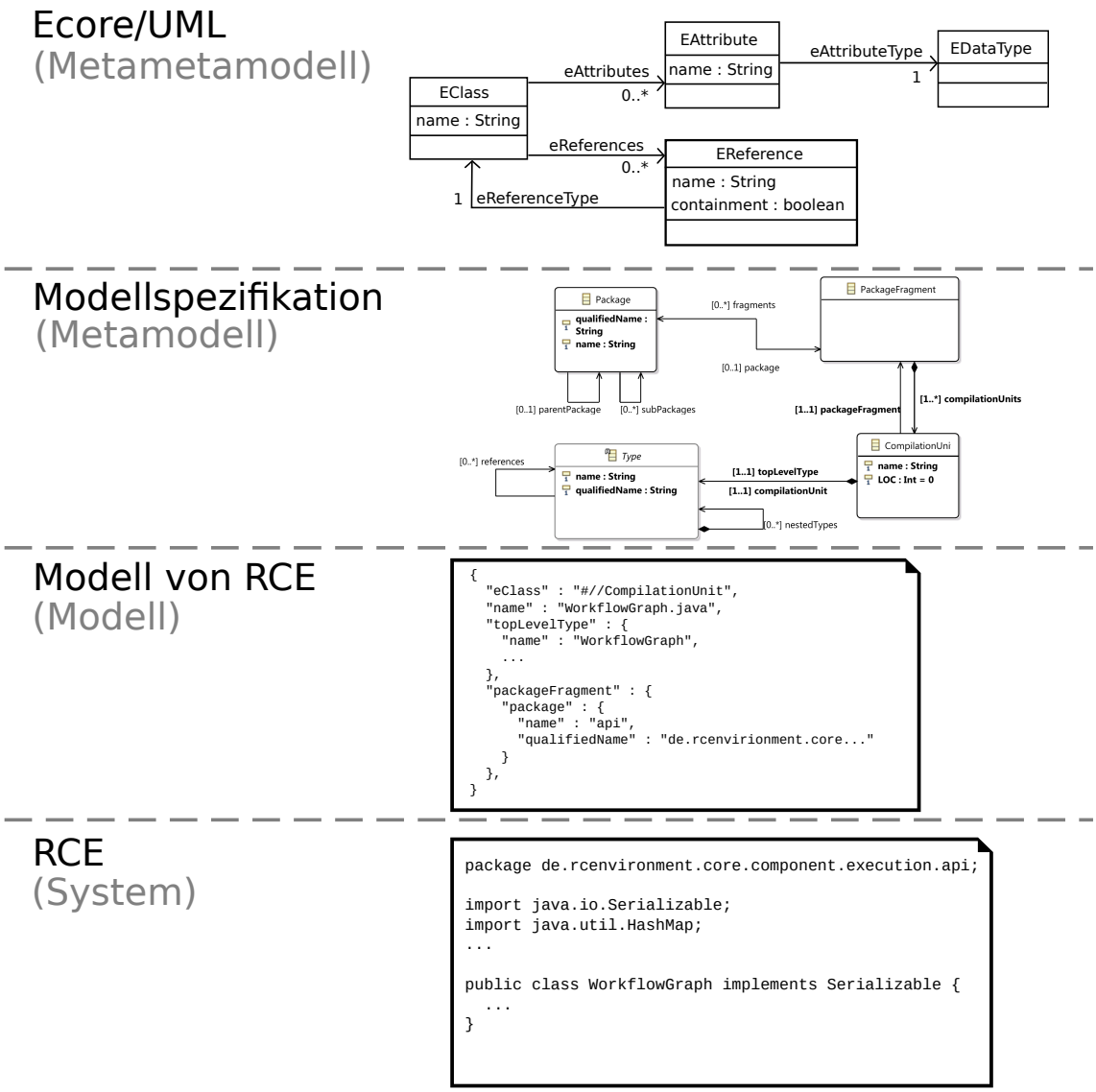


Abbildung 5.3: Modellhierarchie, wie sie bei der Erstellung eines Modells von RCE entsteht.

Abschließend ist zur Übersicht in Abbildung 5.3 die Modellhierarchie dargestellt, die bei der Benutzung von EMF entsteht.

5.1.2 Aufbau

Im Folgenden wird der Aufbau des Metamodells erläutert, welches in Abbildung 5.4. in der Gesamtansicht zu sehen ist. Es lässt sich grob in drei Bereiche einteilen, die sich durch die zugrunde liegenden Datenquellen ergeben:

Java: Packages, Klassen und Methoden

OSGi: Bundles, Services und versionierte Packages

Historie: Commits und Autoren

In allen drei Bereichen musste abgewägt werden, wie umfassend und feingranular das Modell sein soll. Vor allem eine komplette Abbildung der Historie mit allen Details ist nicht praktikabel. Bei den Konzepten von OSGi handelt es sich außerdem um eine statische Sicht auf die OSGi-Anwendung. Die dynamischen Eigenschaften von OSGi können nicht berücksichtigt werden. Das bedeutet zum Beispiel, es wird davon ausgegangen, dass alle Bundles zu Beginn geladen und alle Services direkt registriert werden. Aufbauend darauf, werden die Abhängigkeiten zwischen Bundles und Services durch statische Analyse aufgelöst (mehr dazu in Abschnitt 5.2).

Klassen und Methoden

Ein Teil des Modells bildet den Java-Quellcode ab. Darin enthalten sind, bis zu einem bestimmten Detailgrad, die Abstraktionen und Sprachelemente, die Java in Version 7 besitzt. Der Fokus liegt darauf, genug Daten einzubeziehen, um eine Visualisierung mindestens bis auf Klassenebene möglich zu machen. Dafür gibt es im Modell eine abstrakte Entität *Type*, von der die einzelnen Klassenarten, wie Interfaces, abstrakte Klassen etc. abgeleitet sind (Abbildung 5.5).

Um Metriken für Klassen berechnen zu können, sind Informationen über deren Inhalt nötig. Deshalb beschreibt das Modell auch die Methoden und Instanzvariablen, inklusive deren Bezeichner, Sichtbarkeits- und anderer Modifier. Zudem wird zwischen normalen Methoden, abstrakten Methoden und Konstruktoren unterschieden (Abbildung 5.6).

Generell ist das Design des Modells darauf ausgerichtet, die Struktur einer Software abzubilden, aber keine Metriken zu enthalten. Stattdessen soll die Visualisierung, die das Modell nutzt, die Freiheit haben, alle nötigen Metriken anhand der Struktur selber zu berechnen. Eine Ausnahme bilden die Entitäten für Methoden und Konstruktoren. Deren Größe ist bei der Berechnung von Komplexitätsmetriken unverzichtbar, gleichzeitig soll das Modell aber keine Daten über einzelne Instruktionen oder Konstrukte, wie `for`-Schleifen und `if`-Verzweigungen, enthalten. Deshalb wurde an dieser Stelle ein Attribut für eine *Lines-Of-Code*-Metrik hinzugefügt, das beispielsweise die Anzahl von Instruktionen einer Methode enthält. Welche LOC-Metrik (vgl. Abschnitt 2.6.1) benutzt wird, ist im Modell nicht näher spezifiziert.

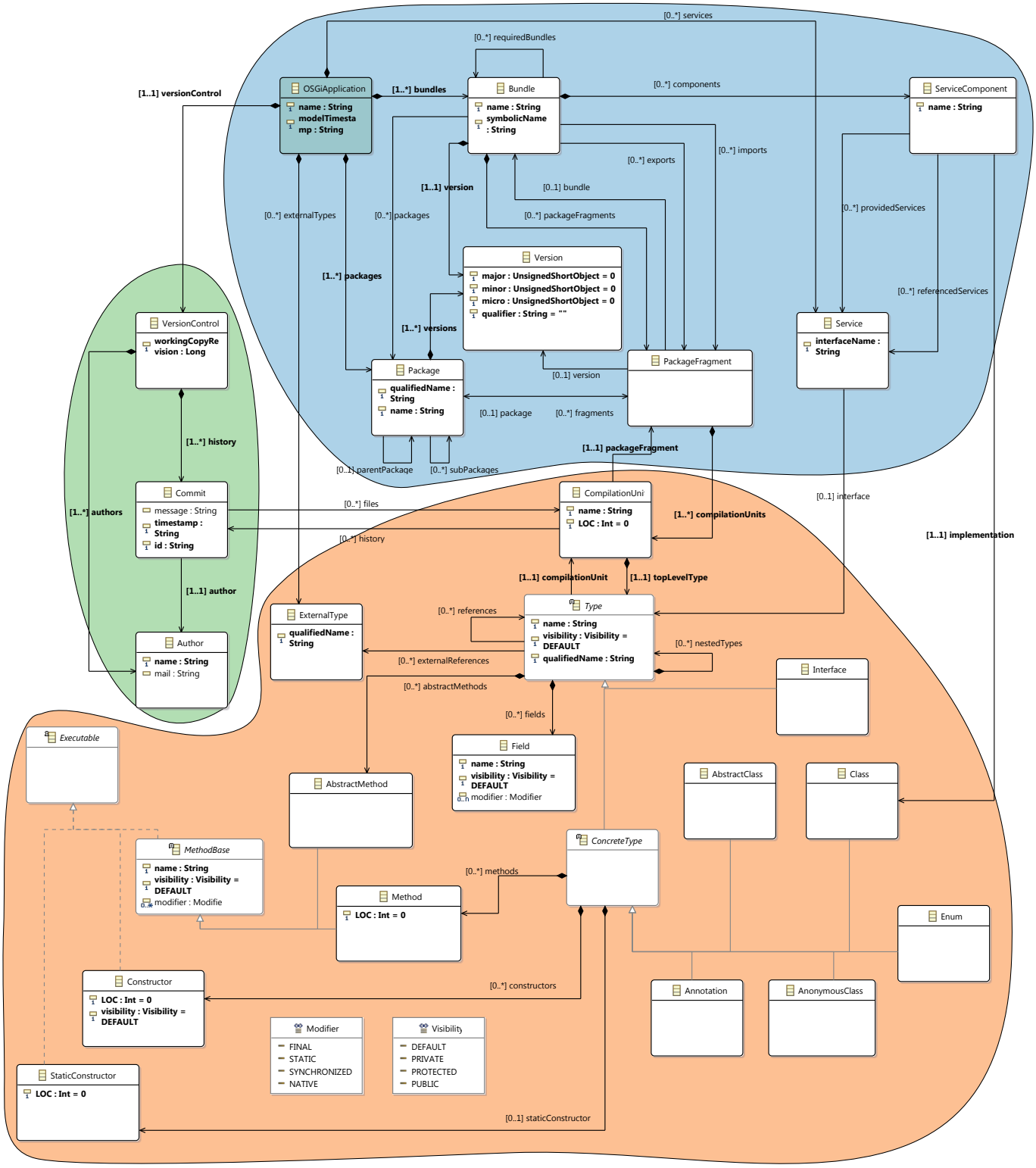


Abbildung 5.4: Das komplette Metamodell, welches sich in drei Bereiche einteilen lässt: OSGi (blau), Java (orange) und Historie (grün).

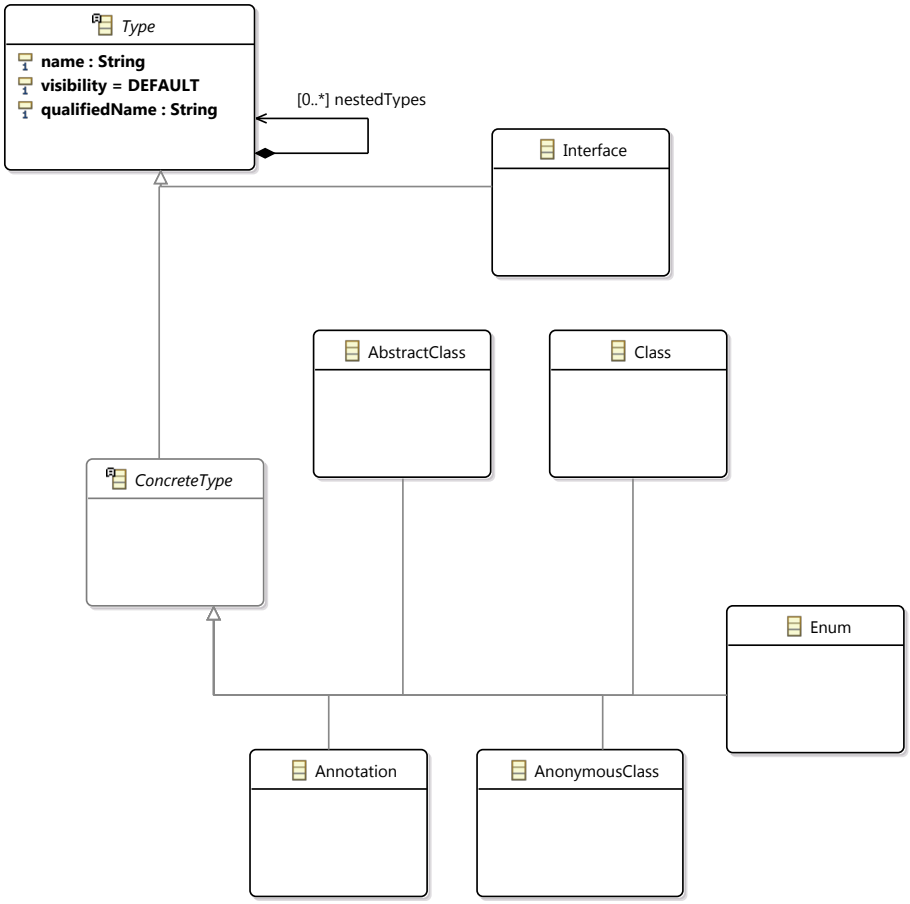


Abbildung 5.5: Die verschiedenen Arten von Klassen im Metamodell.

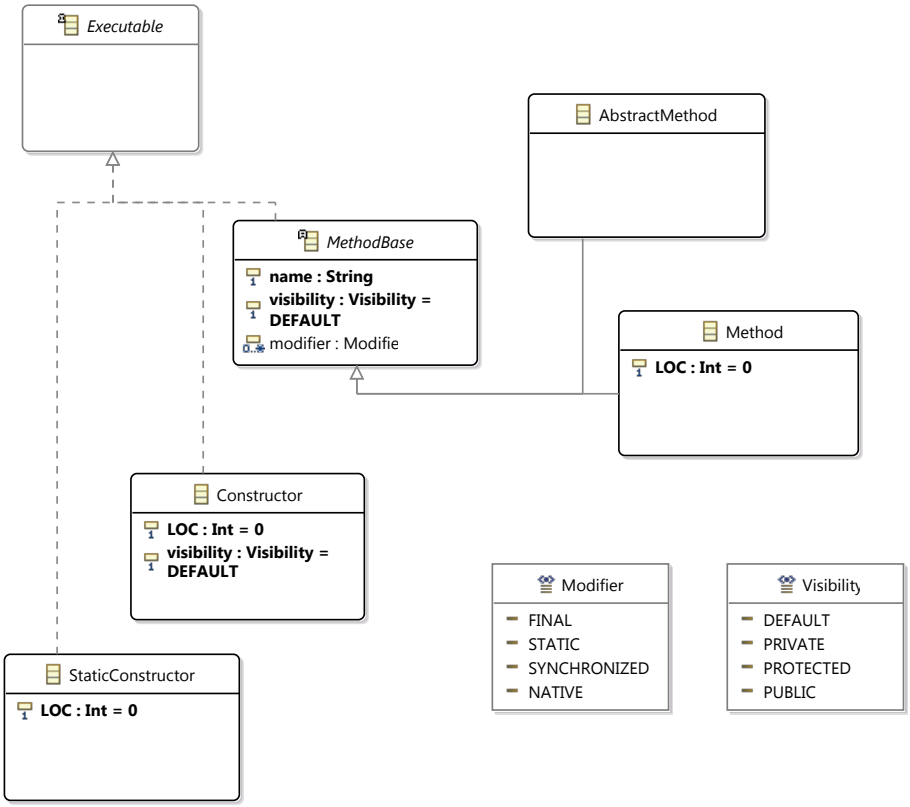


Abbildung 5.6: Verschiedene Arten von Prozeduren im Metamodell.

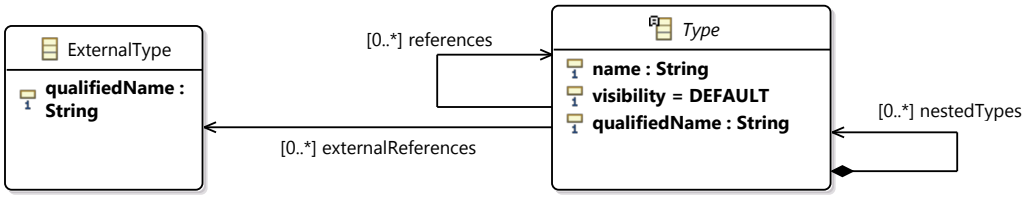


Abbildung 5.7: Referenzen zwischen Typen im Metamodell.

Dies kann also, je nachdem welche Möglichkeiten man bei der Analyse hat, variiert werden.

Desweiteren sind auch die Abhängigkeitsbeziehungen zwischen Klassen im Modell enthalten. Dazu besitzt die Entität *Type*, wie in Abbildung 5.7 dargestellt, verschiedene Referenzen. Eine Referenz von einem Typ A auf einen Typ B bedeutet, dass B von A benutzt wird. Dies schließt alle möglichen Arten von Benutzung ein: Instanzierung, Methodenaufruf, Variablendeklaration, Methodenparameter etc. Um welche Art es sich handelt, wird im Modell allerdings nicht festgehalten.

Somit ist die niedrigste Granularität von Abhängigkeiten im Modell die Klasse. Dadurch hat man zwar die Informationen, dass B von A benutzt wird, aber beispielsweise nicht, welche Methode von B durch A aufgerufen wird. Ebenso wie der Verzicht auf einzelne Instruktionen, ist diese Einschränkung nötig, um die Größe des Modells und damit den Analyseaufwand in praktischen Grenzen zu halten.

Abhängigkeiten auf projektexterne Klassen werden separat behandelt, indem für

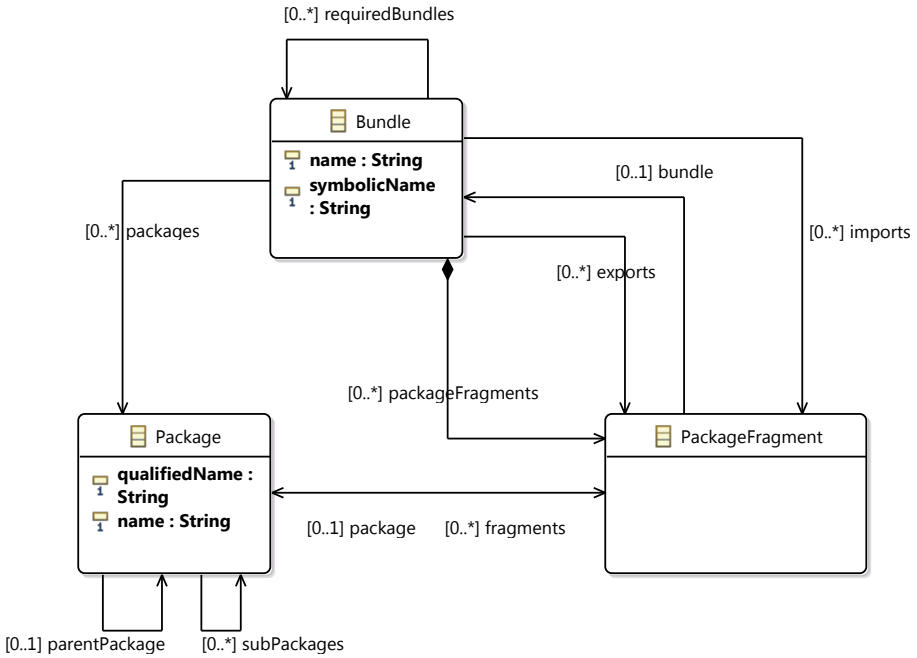


Abbildung 5.8: Unterscheidung zwischen Packages und Package-Fragmenten im Metamodell.

diese eine Entität *ExternalType* erstellt wird. Dies Unterscheidung ist nötig, da über solche Klassen in der Regel keinerlei Informationen, bis auf den vollqualifizierten Klassennamen, bekannt sind. Trotzdem müssen sie im Modell abgebildet werden, um zum Beispiel ermitteln zu können, wo im Code welche externen Bibliotheken verwendet werden.

Bundles und Packages

OSGi-Bundles werden im Modell durch eine eigene Entität *Bundle* repräsentiert. Ein Bundle beinhaltet ein oder mehrere *Packages*, die wiederum *CompilationUnits* mit Klassen beinhalten. Hier entsteht die Verbindung zum oben beschrieben Teil des Modells.

Die Modellierung der Bundle-Struktur stellte eine Herausforderung dar, weil jedes Bundle, wie in Kapitel 2.2 erläutert, seinen eigenen Classloader besitzt. Das bedeutet, ein bestimmtes Package kann gleichzeitig in mehreren Bundles vorhanden sein. Das Gleiche gilt für Compilation-Units. Dieser Umstand muss im Modell berücksichtig werden. Gleichzeitig soll es auch möglich sein, in den Daten einfach zu erkennen, dass ein Package über mehrere Bundles aufgeteilt (das heißt *fragmentiert*) ist.

Aus diesem Grund werden die zwei Entitäten *Package* und *PackageFragment* unterschieden (Abbildung 5.8).

Ersteres ist eine eher abstrakte Repräsention eines Packages, das eineindeutig durch einen vollständig qualifizierten Namen (zum Beispiel `com.example.foo`) definiert ist. Diese Package-Entitäten bilden eine Hierarchie. Das heißt, `com.example.foo` hat eine Referenz *parentPackage* auf das Package mit dem Namen `com.example`. Diese Hierarchie, die implizit durch die Ordner-Struktur und die Namenskonventio-

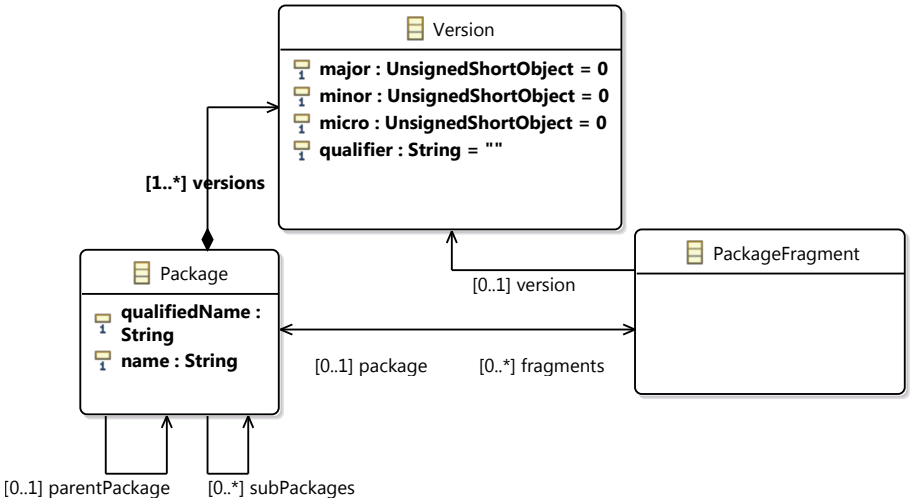


Abbildung 5.9: Packages und deren Versionsnummern im Metamodell.

nen von Packages gegeben ist, hat in Java zwar keine Bedeutung, ist aber bei der späteren Visualisierung nützlich. Streng genommen entspricht die Entität *Package* daher nicht einem Java-Package, weil dieses nur dann vorhanden ist, wenn der entsprechende Ordner auch Compilation-Units enthält.

Diese Rolle übernimmt hingegen die Entität *PackageFragment*. Sie stellt die konkrete Ausprägung eines Packages in einem Bundle dar und enthält mindestens eine Compilation-Unit.

Zwischen *Package* und *PackageFragment* besteht eine bidirektionale Beziehung: Ist ein Package in nur einem Bundle vorhanden und enthält ein oder mehrere Compilation-Units, verweist es auf genau ein Package-Fragment. Existiert es aber in mehreren Bundles (das heißt, es ist fragmentiert), so verweist es entsprechend auf mehrere Package-Fragmente.

Zusätzlich zu dieser Fragmentierung können Package-Fragmente auch eine Version haben, die im Bundle-Manifest bei deren Export optional angegeben werden kann (Abbildung 5.9). Um ermitteln zu können, in welchen Versionen ein Package vorliegt, verweist *Package* auf eine beliebige Anzahl an Versionen, die jeweils einem *PackageFragment* zugeordnet sind.

Services

Das Service-Konzept von OSGi wird durch die zwei Entitäten *Service* und *ServiceComponent* modelliert (Abbildung 5.10).

Ein Service ist letztendlich nur eine Klasse, die von einem Bundle als Service angeboten wird. Üblicherweise handelt es sich dabei um ein Java-Interface, was aber nicht vorgeschrieben ist. Aus diesem Grund verweist *Service* auf einen *Type*.

Ein *ServiceComponent* repräsentiert eine Service-Deklaration durch eine XML-Datei in einem Bundle. Ein *ServiceComponent* kann Services zur Verfügung stellen (*provide*) oder benutzen (*reference*). Außerdem muss jedes ServiceComponent durch eine bestimmte Klasse implementiert sein.

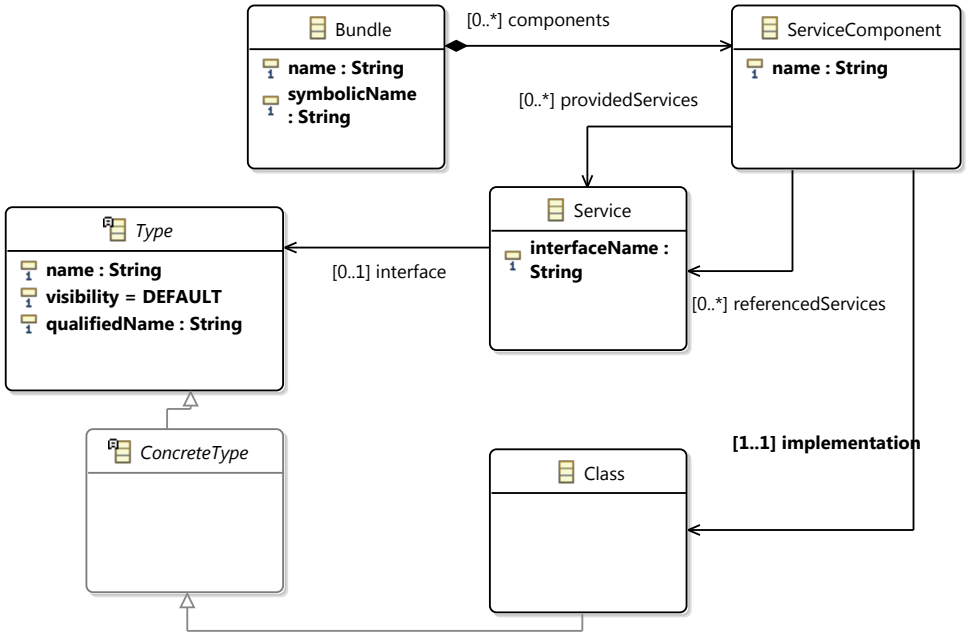


Abbildung 5.10: Services und Service-Components im Metamodell.

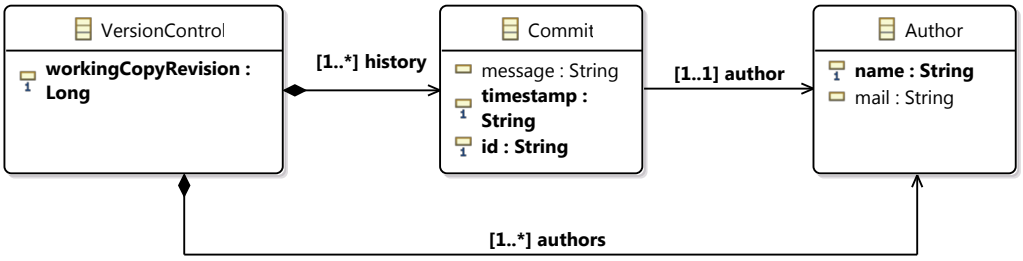


Abbildung 5.11: Historie aus der Versionsverwaltung im Metamodell.

Diese Elemente reichen aus, um alle Aspekte von deklarativen Services zu modellieren.

Historie

Betrachtet man die Historie einer Software, gewinnt das Datenmodell deutlich an Größe. Im Prinzip entsteht durch die Zeitachse eine dritte Dimension in den Daten, da sich alle Entitäten des Modells über die Zeit verändern können. Die komplette Historie im Modell abzubilden, wurde, wegen der großen Datenmenge die dabei entsteht, ausgeschlossen. Das Modell enthält nur historische Informationen über die Quellcodedateien die in der aktuellen Revision vorhanden sind. Ein „Rollback“ zu einer vorherigen Revision ist anhand der Daten aber nicht möglich.

Das Versionskontrollsystem wird durch die drei Elemente *VersionControl*, *Commit* und *Author* modelliert (Abbildung 5.11). Es stehen also die Informationen zur Verfügung, die beispielsweise durch `svn log` ausgegeben werden.

Versionskontrollsysteme arbeiten auf Dateiebene, also im Fall von Java-Quellcode, mit Compilation-Units. Deshalb besteht eine (bidirektionale) Beziehung zwischen

Commits und Compilation-Units. So ist es möglich, alle Commits zu ermitteln in denen eine bestimmte Compilation-Unit verändert wurde. Außerdem können ausgehend von einem Commit die zugehörigen Compilation-Units bestimmt werden. Wie schon erwähnt, werden dabei aber nur die Dateien berücksichtigt, die in der aktuellen Revision existieren. Informationen über gelöschte Dateien werden nicht einbezogen.

Bei der Modellierung wurde darauf geachtet, das Modell unabhängig vom verwendeten Versionskontrollsystem zu halten. Deshalb kann die `id` eines Commits sowohl eine Subversion-Revisionsnummer als auch einen Commit-Hash aus git darstellen.

5.1.3 Mögliche Erweiterungen

Das vorliegende Metamodell bietet eine gute Grundlage für die Analyse und Visualisierung. Es sind alle Daten enthalten, um das Konzept, wie im vorherigen Kapitel beschrieben, umsetzen zu können.

Trotzdem gibt es viel Potential für Erweiterungen. Besonders der Teil zur Historie könnte ausgebaut werden. Momentan haben nur die Java Compilation-Units eine Verbindung zur Commit-Historie. Doch auch Bundle-Manifeste und Servicedeklarationen können sich über die Zeit ändern.

Desweiteren könnte die Granularität bei der Modellierung des Quellcodes verfeinert werden, um Daten auf Instruktionsebene mit einzubeziehen.

Im Bereich von OSGi wurden einige Konzepte komplett außen vor gelassen (siehe Abschnitt 5.2.7). Auch hier gibt es also Erweiterungsmöglichkeiten.

5.2 Extraktion und Analyse

Wie in Abschnitt 4.6 erläutert, werden im ersten Teil der Implementierung Daten aus einem OSGi-Projekt extrahiert, analysiert und in das Modell transformiert.

Als Programmiersprache wurde hierfür Java SE 7 ausgewählt. Entscheidendes Kriterium für diese Wahl waren die verfügbaren Bibliotheken. Vor allem bei der Datenextraktion aus Quellcode und Versionsverwaltung sollen diese zum Einsatz kommen, da eine eigene Implementierung solcher Funktionen nicht zielführend wäre.

Auch andere Sprachen, wie C++ oder Python, kämen in Frage, aber besonders für die Analyse von Java-Code erscheinen diverse, in Java geschriebene, Bibliotheken als am besten geeignet. Zudem ermöglicht erst die Implementierung in Java die sinnvolle Nutzung des Eclipse Modeling Framework, wie sie im vorherigen Abschnitt beschrieben wurde.

5.2.1 Datenextraktion

Zunächst müssen die benötigten Daten aus den zur Verfügung stehenden Informationsquellen extrahiert werden. Dafür wurden verschiedene Java-Bibliotheken evaluiert und ausgewählt.

Quellcode und Package-Struktur

Die Konzepte von Java, wie Packages, Klassen und Methoden, machen einen großen Teil des Metamodells aus. Gleichzeitig ist deren Analyse vergleichsweise komplex, da dazu Java-Code geparst werden muss. Während das Auslesen von Klassennamen und Methodensignaturen noch vergleichsweise einfach ist, ist das Ermitteln von Abhängigkeiten zwischen Klassen schon deutlich schwieriger. Wie in Abschnitt 2.3.1 erläutert, können Abhängigkeiten auf viele verschiedene Arten entstehen. Das Auslesen von Import-Statements reicht nicht aus. Deshalb ist die Auswahl einer geeigneten Analyse-Bibliothek hier besonders wichtig.

Eine solche ist etwa über die Java Development Tools (JDT) aus dem Eclipse-Projekt [88] verfügbar. Die sogenannte *JDT Core* API stellt eine Schnittstelle zur Analyse von Java-Code bereit, die von der Eclipse-IDE für Zwecke wie Autovervollständigung, Fehleranalyse, Refactoring und das Anzeigen der Code-Struktur benutzt wird [95].

JDT Core enthält einen Java-Compiler und ein eigenes Metamodell. Metamodelle von Compilern sind meist auf Code-Optimierung und effiziente Kompilierung ausgelegt. Hauptzweck des JDT-Metamodells hingegen ist es, im Rahmen der Eclipse IDE, die oben genannten Aufgaben möglichst gut zu unterstützen.

JDT Core lässt sich auch außerhalb der Eclipse IDE als Bibliothek nutzen. Nachteile sind aber die vielen Abhängigkeiten zu anderen Bibliotheken des Eclipse-Projekts. Desweiteren ist die Dokumentation von Metamodell und API nicht besonders umfangreich. So ist beispielsweise nicht dokumentiert, wie JDT Core zur Analyse von Quellcode benutzt werden kann, der nicht als Projekt in einem Eclipse-Workspace vorliegt.

Deshalb wurde eine andere Bibliothek namens *Spoon* [102] gewählt. Diese verwendet intern zwar den JDT-Compiler, stellt aber eine einfachere API und ein eigenes Metamodell zur Verfügung. Spoon ist eine Bibliothek, die es ermöglicht, mit geringem Aufwand domänenspezfische Analyse und Transformation von Java-Code zu schreiben, ohne sich mit Parsing und anderen Compilertechniken auseinandersetzen zu müssen [58].

Abbildung 5.12 stellt einen vereinfachten Ausschnitt des Metamodells von Spoon dar, welches die Struktur von Java-Programmen abbildet. Hier sind Ähnlichkeiten zu dem im vorherigen Abschnitt entwickelten Metamodell erkennbar. Neben diesen strukturellen Elementen enthält das Metamodell von Spoon sogenannte Code-Elemente, die ausführbaren Java-Code beschreiben und sich in *Statements* und *Expressions* unterteilen lassen (Abbildung 5.13).

Eine wichtige Eigenschaft von Spoon ist, dass es schwache Referenzen (Weak References) benutzt, um Beziehungen zwischen Modellelementen darzustellen [58]. Das bedeutet, bei der Analyse wird versucht, Referenzen zwischen Elementen aufzulösen. Ist dies aber nicht möglich, weil sich das referenzierte Element nicht im Modell befindet, verweist die Referenz nur auf eine textuelle Beschreibung des Elements, die sich aus der Referenz selbst ergibt. Das ist beispielsweise nützlich, wenn eine Klasse referenziert wird, die sich in einer externen Bibliothek befindet, welche nicht in die Analyse einbezogen wird.

In diesem Fall besitzt die referenzierende Klasse ein Objekt vom Typ `CtTypeRefe-`

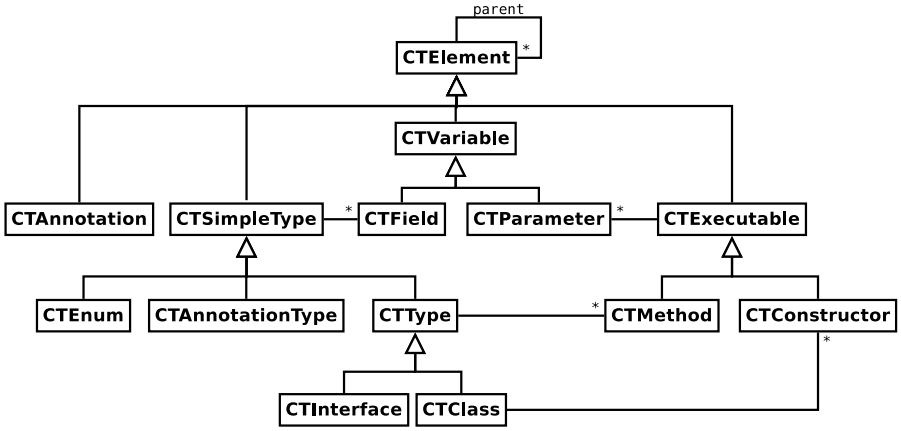


Abbildung 5.12: Ausschnitt des Spoon-Metamodells, welcher die Struktur von Java-Code darstellt (Quelle: [58]).

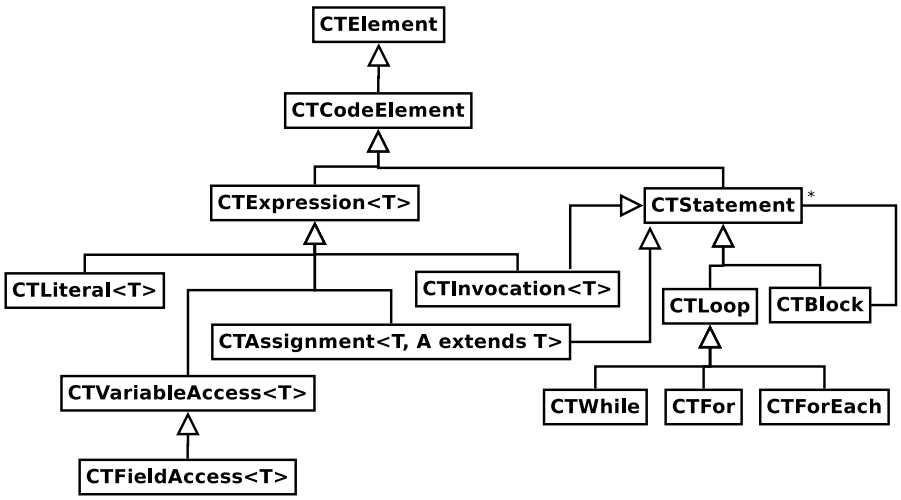


Abbildung 5.13: Ausschnitt des Spoon-Metamodells, welcher die Ausdrücke und Instruktionen in Java darstellt (Quelle: [58]).

`rence`, die aber nicht auf ein `CtClass`-Objekt verweist, sondern nur einen `String` mit dem Namen der Klasse enthält. Wie im Folgenden zu sehen sein wird, ist dieses Verhalten besonders bei der Analyse von OSGi-Software von großem Nutzen.

Manifest-Dateien

Da die Manifest-Dateien in OSGi-Bundles in einem einfachen Key-Value-Format geschrieben sind, ist deren Analyse vergleichsweise einfach. Die Java-Standardbibliothek bietet die Klasse `java.util.jar.Manifest` an, um auf Manifestdateien zuzugreifen [96] . Diese gibt allerdings für jeden Value nur einen String zurück. Da in OSGi die Werte aber oft ein komplexeres Format aus mehreren zusammengesetzten Werten und Attributen haben (siehe Abschnitt 2.2), wurde stattdessen eine Utility-Klasse aus dem Equinox-Framework dafür benutzt [97].

Service-Deklarationen

Zum Lesen von XML-Dateien bietet die Standard-Bilbiothek von Java mehrere APIs an (DOM, SAX, Stax, JAXB) [77]. Der Einfachheit halber wird hier die DOM[2] API verwendet. Sie ermöglicht einen wahlfreien Zugriff auf einzelne XML-Elemente, weil das komplette Dokument dabei in Form eines DOM-Baums im Arbeitspeicher liegt. Da die Service-Deklarationen bei OSGi in der Regel sehr kurz sind, ist der Speicherbedarf hier nicht von Nachteil.

Versionshistorie

Da das RCE-Projekt, an dem die vorliegende Arbeit evaluiert wird, Subversion als Versionskontrollsystem einsetzt, wurde zunächst nur die Analyse von Subversion-Repositories implementiert. Es gibt mehrere Java-Bibliotheken, die dafür APIs bereitstellen und in Erwägung gezogen wurden:

- JavaHL
- SVNClientAdapter
- SVNKit

JavaHL ist Teil des Subversion-Projekts und das offizielle Java-Binding an die Subversion API [94]. Diese ist nativ in C implementiert, weshalb für die Verbindung zu Java das *Java Native Interface* (JNI) nötig ist. JNI ermöglicht es in C oder C++ geschriebene APIs in einer Java-Anwendung zu benutzen. JavaHL bietet einen umfangreichen Zugriff auf Subversion über eine Low-Level API, die sowohl vom Subversion-Komandozeilenclient als auch diversen grafischen Clients verwendet wird. Für die einfache Datenextraktion aus einem Repository ist jedoch eine einfachere Schnittstelle auf höherer Ebene wünschenswert.

So eine Schnittstelle bietet SVNClientAdapter, welches aber eine Low-Level-API benötigt, auf der es aufsetzen kann [103]. Dies kann entweder JavaHL oder das

[2] *Document Object Model*

```
1  class PackageProcessor extends AbstractProcessor<CtPackage> {
2
3    // A bundle-entity of our model
4    Bundle bundle;
5
6    @Override
7    public void process(CtPackage spPkg) {
8      Package pkg = new Package();
9      Package.setQualifiedName(spPkg.getQualifiedName());
10     bundle.addPackage(pkg);
11   }
12 }
```

Listing 5.1: Einfacher *Processor*, der eine Modelltransformation ausführt.

Kommandozeilen-Interface von Subversion sein. Beides erfordert das Ausführen von nativem Code, was zu Plattformabhängigkeit führt. Da die Analyse aber ohne die Installation zusätzlicher nativer Bibliotheken oder Programme auskommen sollte, wurde stattdessen SVNKit als Bibliothek für Subversion ausgewählt.

SVNKit enthält ein komplett in Java implementiertes Backend [104]. Es bietet sowohl eine High-Level-API zur Verwaltung von Arbeitskopien als auch eine Low-Level-API zum direkten Zugriff auf Subversion Repositories an. Im Rahmen dieser Arbeit kommt erstere zum Einsatz. Sie orientiert sich an den üblichen Subversion-Kommandos wie `status`, `log` und `info`.

5.2.2 Modelltransformation

Nach der Extraktion der Daten muss mit diesen ein Modell erstellt werden, dessen Struktur durch das Metamodell in Abschnitt 5.1 definiert ist. Da Bibliotheken wie Spoon ebenfalls ein eigenes Datenmodell besitzen, wird dieser Vorgang auch *Modelltransformation* genannt.

Die API von Spoon bietet mehrere Möglichkeiten an, auf die Daten in dessen Modell zuzugreifen. Um bestimmte Elemente im Modell programmatisch abzufragen, können *Queries* aus sogenannten *Filtern* formuliert werden. Eine Alternative, von der in dieser Arbeit Gebrauch gemacht wurde, sind *Processors*. Diese implementieren ein Visitor-Pattern, um Modell-Elemente eines bestimmten Typs analysieren zu können.

Dazu muss der Benutzer der API nur eine *process*-Methode schreiben, welche automatisch die zu analysierenden Elemente einzeln übergeben bekommt. In Listing 5.1 ist vereinfachter Code zu sehen, welcher alle Packages eines Bundles aus dem Spoon-Modell in unser Metamodell transformiert.

5.2.3 Serialisierung und Persistierung

Wie in Abschnitt 4.6 erläutert, sind Analyse und Visualisierung zwei unabhängige Anwendungen zwischen denen ein Austausch des Datenmodells stattfinden muss.

EMF beinhaltet ein Persistierungs-Framework, das standardmäßig das Serialisieren und Speichern von Ecore Modellen als XMI (*XML Metadata Interchange*) unterstützt [71]. Bei XMI handelt es sich um einen Standard für ein Format, dass den Austausch von Modellen (wie zum Beispiel UML und Ecore) zwischen Anwendungen erlaubt. Zusätzlich dazu gibt es viele Erweiterungen die anspruchsvolleren Anforderungen gerecht werden, aber natürlich auch deutlich mehr Komplexität für die Anwendung mit sich bringen. Sie erlauben zum Beispiel Versionierung, hohe Skalierbarkeit, Client-Server-Architektur, Mehrbenutzerzugriff und Transaktionen. Beispiele für solche Backends sind: CDO, EMFStore und Teneo [89]. Neben der Persistierung in Dateien erlauben diese Systeme auch den Einsatz von relationalen oder objektrelationalen Datenbanken

Die Wahl der Persistierungsart hängt im Rahmen dieser Arbeit vor allem davon ab, in welcher Programmiersprache die Visualisierung umgesetzt wird. Sie muss in der Lage, sein das Modell ohne großen Implementierungsaufwand und performant zu lesen. Standardmäßig ist EMF auf Java-Anwendungen ausgerichtet, aber es gibt auch Persistierungsmöglichkeiten, die kompatibel zu anderen Programmiersprachen sind.

Wie in Abschnitt 5.3 noch näher erläutert wird, ist die Visualisierung in JavaScript implementiert. Ein natürliches Serialisierungsformat in JavaScript ist JSON, welches ein menschenlesbares Serialisierungsformat für JavaScript-Objekte ist.

Deshalb fiel die Wahl auf EMFJson [90]. Mit diesem gibt es eine einfache Möglichkeit, Ecore-Modelle in JSON-Dateien zu speichern. Die Integration von EMFJson bedeutet sowohl auf Analyse- als auch auf Visualisierungsseite kaum Mehraufwand. Es war aber zunächst unklar, ob dieser Weg bei sehr großen Modellen performant genug ist, da das Modell in einer Textdatei gespeichert wird und von JavaScript komplett in den Speicher geladen wird. Deshalb wurde auch die Nutzung einer Datenbank in Erwägung gezogen. Wie sich aber herausstellte, ergeben sich durch JSON keine Performance-Probleme (mehr dazu in Kapitel 6).

5.2.4 Konfiguration

Da die Ordner-Struktur von OSGi-Projekten sehr verschieden sein kann, kann diese bei der Analyse nicht komplett automatisiert erfasst werden. Deshalb muss der Benutzer für jedes zu analysierende Projekt einige Angaben in einer Konfigurationsdatei machen. Eine Beispielkonfiguration ist im Anhang zu finden.

Darin wird angegeben, wo die Bundles liegen und welche Bundles bei der Analyse berücksichtigt oder ausgeschlossen werden sollen. Desweiteren muss dem Projekt ein Name gegeben und der Speicherort für das Modell bestimmt werden. Um die Historie bei der Analyse mit einzubeziehen, müssen der Wurzelfpad der Arbeitskopie sowie die Zugangsdaten des Subversion-Repositories angegeben werden. Im aktuellen Stand der Implementierung wird die Authentifizierung via Benutzername und Passwort sowie mit einem (passwortgeschützten) SSH-Key-File unterstützt.

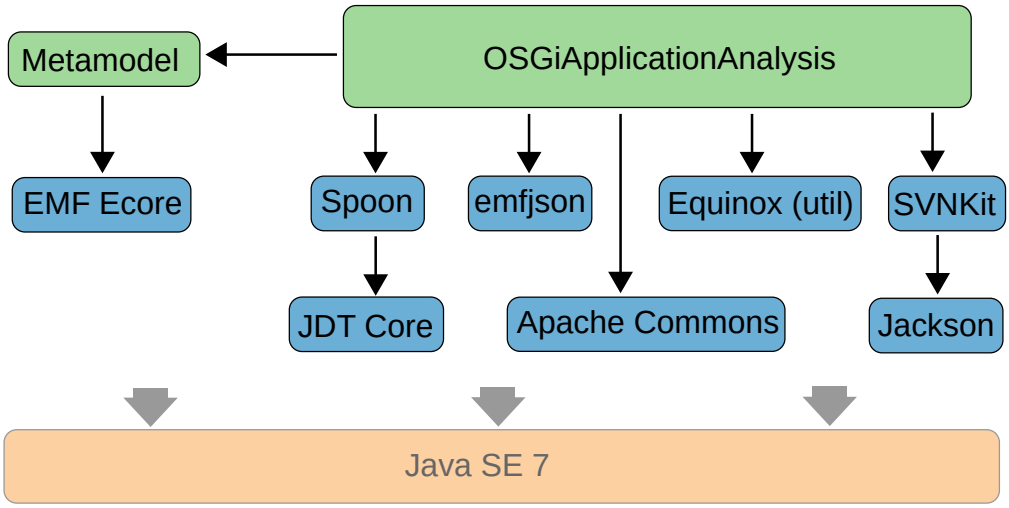


Abbildung 5.14: Abhängigkeiten des Analyse-Codes.

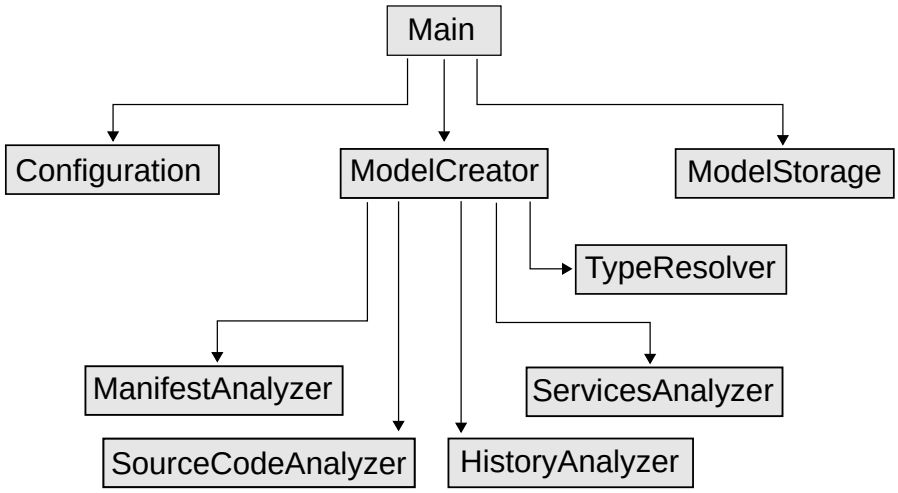


Abbildung 5.15: Überblick über zentrale Klassen des Analyse-Codes.

5.2.5 Architektur

In Abbildung 5.14 ist eine Übersicht über die Abhängigkeiten der implementierten Anwendung zu sehen. Diese wird *OSGiApplicationAnalysis* genannt und benutzt zur Datenextraktion die in Abschnitt 5.2.1 erwähnten Bibliotheken (Spoon, SVNKit, emfjson und Equinox) sowie die Apache Commons. Zusätzlich besteht eine Abhängigkeit zu den Klassen des Metamodells, welche mit EMF generiert werden.

Der interne Aufbau von OSGiApplicationAnalysis ist in Abbildung 5.15 als Klassendiagramm skizziert. Dabei handelt es sich um einen Ausschnitt mit den wichtigsten Klassen. Einstiegspunkt in den Programmablauf ist die Klasse `Main`, die das Lesen der Konfigurationsdatei, das Erstellen des Modells und das Speichern des Modells über die jeweiligen Klassen anstößt.

`ModelCreator` ist dafür zuständig, das komplette Modell zusammenzutragen, wofür verschiedene *Analyzer* benutzt werden. Diese sind jeweils für die Erstellung eines

Teiles des Modells verantwortlich:

ManifestAnalyzer Liest Manifest-Dateien von OSGi-Bundles mit Equinox Utility-Klassen.

SourceCodeAnalyzer Analysiert den Java-Quellcode mit Spoon.

ServiceAnalyzer Liest die Service-Deklarationen über die DOM-API von Java.

HistoryAnalyzer Extrahiert Daten aus Subversion mit SVNKit.

Die Analyzer erzeugen die Elemente des Modells für die sie zuständig sind und geben diese an den `ModelCreator` zurück. Er fügt die einzelnen Teile zum kompletten Modell zusammen.

Die Aufteilung des Codes auf verschiedene Analyzer soll die Erweiterung um weitere Datenquellen und Analysebibliotheken erleichtern. So wäre es zum Beispiel denkbar einen weiteren Analyzer für die Analyse von Testergebnissen zu integrieren. Dadurch, dass die Analyzer jeweils bestimmte externe Bibliotheken kapseln, ist deren Austausch jederzeit möglich. Beispielsweise ließe sich im SourceAnalyzer Spoon durch JDT ersetzen. Ebenso könnte ein zweiter HistoryAnalyzer geschrieben werden, der SVNKit durch eine entsprechende Anbindung an git-Repositories ersetzt. Solche Änderungen können ohne Anpassung der Analyzer-Schnittstellen vorgenommen werden, so dass eine Änderung von ModelCreator oder anderen Klassen nicht nötig wäre.

5.2.6 Herausforderungen

Bei der Implementierung der Analyse gab es durch die Besonderheiten von OSGi-Anwendungen diverse Herausforderungen.

Statische Bestimmung der Bundle-Abhängigkeiten

Abhängigkeiten zwischen Bundles werden von OSGi normalerweise erst zur Laufzeit behandelt [82]. Um diese Abhängigkeiten im Modell abbilden zu können (siehe auch Abschnitt 5.1.2), müssen sie schon während der Analyse aufgelöst werden. Aufgabe dabei ist es, für jedes importierte Package eines Bundles, das passende exportierte Package eines anderen Bundles zu finden. Dabei müssen nicht nur die optionalen Versionsangaben, sondern auch einige Sonderfälle, wie zum Beispiel die Kollision von einer *Import-Package*-Anweisung mit einer *RequireBundle*-Anweisung beachtet werden.

OSGi-Classloading und Spoon

Wie in Abschnitt 2.2.2 erläutert, verwendet in OSGi jedes Bundle seinen eigenen Classloader, was im Gegensatz zu normalen Java-Anwendungen dazu führt, dass mehrere Klassen mit dem selben Namen existieren können. Das Problem an dieser Stelle ist, dass Spoon bei der Analyse von Source-Code den Standard-Classloader der JVM benutzt, um Klassen des analysierten Codes zu laden und Referenzen zwischen

Klassen aufzulösen. Bei der Analyse eines kompletten OSGi-Projekts schlägt das natürlich fehl, da ein solches nicht durch nur einen einzigen Classloader geladen werden kann. Grundsätzlich bieten sich zwei Lösungen für dieses Dilemma an:

1. Implementierung eines eigenen Classloaders für Spoon, der das Verhalten der OSGi-Classloader imitiert und für jedes Bundle einen separaten Namespace für Klassen ermöglicht.

2. Jedes Bundle separat mit Spoon analysieren.

Die erste Variante wurde verworfen, da sie einen großen Mehraufwand bei der Realisierung bedeuten würde. Stattdessen wird jedes Bundle einzeln mit Spoon analysiert. Dies hat zur Folge, dass Referenzen zwischen Elementen über Bundlegrenzen hinweg, nicht automatisch von Spoon aufgelöst werden können (vgl. Erklärung zu „schwachen Referenzen“ in Abschnitt 5.2.1). Damit Spoon ein Modell mit schwachen Referenzen zwischen Klassen aufbauen kann, muss explizit dessen *No-Classpath*-Modus aktiviert werden.

Auflösung von Referenzen zwischen Bundles

Durch die Analyse einzelner Bundles und die Benutzung des *No-Classpath*-Modus müssen Referenzen zwischen Klassen[3], die über Bundlegrenzen hinweg verlaufen, selber aufgelöst werden. Das heißt, einer nicht aufgelösten Referenz, die in Spoon nur den vollständig qualifizierten Namen des Typs enthält (zum Beispiel `com.example.foo.Bar`), muss die zugehörige Modell-Entität aus einem anderen Bundle zugeordnet werden. Diese Aufgabe übernimmt der sogenannte `TypeResolver` (siehe Abbildung 5.15).

Er berücksichtigt, welche Packages von der referenzierenden Klasse aus, durch Package-Imports des Bundles, sichtbar sind und bildet den Klassennamen der Referenz auf das entsprechende Objekt ab.

Undefiniertes Verhalten von OSGi

An einigen Stellen wird in der OSGi-Spezifikation nicht genau dokumentiert wie sich OSGi-Implementierungen in bestimmten Situationen verhalten sollen. Wird beispielsweise ein Package in der gleichen Version von zwei Bundles exportiert, ist nicht definiert, welches Package-Fragment beim Import durch ein anderes Bundle tatsächlich benutzt wird. Diese Situation sollte zwar durch den Programmierer verhindert werden, ist aber grundsätzlich möglich und muss bei der Analyse berücksichtig werden. Deshalb wird in diesem Fall vom `TypeResolver` eine entsprechende Warnung in der Log-Datei ausgegeben.

5.2.7 Einschränkungen und Erweiterungsmöglichkeiten

Im aktuellen Stand der Implementierung von OSGiApplicationAnalysis gibt es einige Einschränkungen, die bei der Benutzung beachtet werden müssen.

[3] Im Metamodell: *reference*-Beziehung von *Type* auf sich selbst

Es wurden nur die gängigsten OSGi-Features bei der Analyse berücksichtigt. Viele Eigenschaften von OSGi-Anwendungen, die in der OSGi-Spezifikation [3, 2] beschrieben werden, sind aber nicht Bestandteil des Modells. Tabelle 5.1 gibt dazu eine Übersicht.

Berücksichtigt	Unberücksichtigt
Bundle-SymbolicName	Service-Nutzung über API
Bundle-Name	Service-Deklaration über Annotationen
Bundle-Version	Fragmentierte Bundles
Require-Bundle	Capabilities
Export-package (inkl. Version)	Version von Bundles
Import-package (inkl. Version)	...
XML-Service-Deklaration	

Tabelle 5.1: Übersicht über, bei der Analyse berücksichtigte und unberücksichtigte, OSGi-Features, die im Hinblick auf die Software-Architektur von Interesse sein können.

Weitere Einschränkungen ergeben sich aus der Nutzung des *No-Classpath*-Modus von Spoon. In diesem Modus gibt es einige Situationen in denen kein vollständig qualifizierter Klassenname für eine Typ-Referenz ermittelt werden kann. In diesen Fällen können die Referenzen nicht aufgelöst und damit nicht im Modell berücksichtigt werden:

- Referenzen auf Klassen, die aus einem anderen Bundle über *Star*-Imports importiert werden (zum Beispiel `import com.example.*`)
- Referenzen auf anonyme Klassen.
- Referenzen auf Klassen, die durch Benutzung von Instanzvariablen einer nichtaufgelösten Oberklasse zustande kommen.

Deshalb muss generell beachtet werden, dass Referenzen auf Klassenebene nicht vollständig im Modell enthalten sind.

Desweiteren konnte aus Zeitgründen für Methoden keine Lines-of-Code-Metrik implementiert werden. Das entsprechende Attribut im Modell ist daher immer 0. Die Lines-of-Code von Compilation-Units hingegen werden durch Zählen aller Zeilen, inklusive Leerzeilen und Kommentaren, berechnet. Hier bietet es sich an, eine oder mehrere der in Abschnitt 2.6.1 beschriebenen Metriken, umzusetzen.

Auch an anderen Stellen in der Analyse gibt es Raum für Erweiterungen. Zum Beispiel wäre es wünschenswert, neben Subversion auch andere Versionsverwaltungssysteme (git, Mercurial ...) zu unterstützen. Bei der Analyse der Historie werden Umbenennungen von Dateien noch nicht brücksichtigt. Das kann bei ihrer Betrachtung zu falschen Schlussfolgerungen führen, weil eine Datei nach ihrer Umbennung nur noch auf den letzten Commit verweist und keine Verbindung mehr zu der davor liegenden Historie aufweist.

Weitere Verbesserungen wären zum Beispiel eine grafische Benutzeroberfläche zur Konfiguration der Analyse oder Erweiterung der Authentifizierungsmöglichkeiten für Subversion.

5.3 Visualisierung

Im Folgenden wird die Umsetzung der Visualisierung, wie sie in Abschnitt 4.5 skizziert wurde, vorgestellt.

5.3.1 Technische Umsetzung

Zur Umsetzung der Visualisierung standen drei grundlegende Ansätze zur Auswahl, deren Vor- und Nachteile im Folgenden zusammgengefasst wurden:

Eigenständige Anwendung:
+ Performance
+ keine funktionalen Einschränkungen
+ freie Wahl der Programmiersprache
− Implementierungsaufwand
− Installation durch den Benutzer nötig

Integration in IDE:
+ Integration in bestehenden Workflow
+ Keine zusätzliche Anwendung nötig
+ Verbindung von Visualisierung und Quellcode
− begrenzte Funktionalität durch Plugin-Mechanismus
− Festlegung auf eine IDE

Webbasiert im Browser:
+ plattformunabhängig
+ sowohl lokal als auch serverseitige Anwendung möglich
+ überall vorhanden, keine zusätzliche Installation nötig
− Beschränkung auf Webtechnologien und Javascript
− Performance

Die Wahl fiel auf eine Implementierung mit Webtechnologien im Browser, basierend auf Javascript. Ausschlaggebend für diese Entscheidung waren die Vorteile des webbasierten Ansatzes und die verfügbaren Bibliotheken für Javascript.

Neben Plattformunabhängigkeit und der geringen Einstiegshürde für den Benutzer, bietet eine Visualisierung im Browser die Möglichkeit, die Analyse bei Bedarf im Hintergrund auf einem Server auszuführen. Denkbar ist auch eine Integration der Analyse in einen *Continuous-Integration*-Prozess, so dass die über den Server aufrufbare Visualisierung immer den aktuellen Stand der Software widerspiegelt.

Wie in Abschnitt 5.2.3 beschrieben, können die extrahierten Daten nach der Analyse mit EMFJson als JSON-Datei serialisiert werden, was ein einfaches Laden mit

Javascript möglich macht.

Die eigentliche Datenvisualisierung wurde mit Hilfe der Javascript Bibliothek *D3* umgesetzt [86]. *D3* steht für *Data-Driven Documents*, was den Zweck der Bibliothek sehr allgemein beschreibt. Sie bietet eine umfangreiche API, um Dokumente, üblicherweise in Form von HTML und SVG, auf der Grundlage von Daten zu manipulieren.

Die Hauptfunktionen von D3 sind [49]:

- *Laden* von Daten im Browser.
- *Binden* von Daten an Elemente des Dokuments und Erzeugung neuer Elemente im Dokument anhand der Daten.
- *Transformation* der Elemente und Manipulieren ihrer visuellen Eigenschaften auf Grundlage der gebundenen Daten.
- *Übergang* der Elemente zwischen verschiedenen Zuständen basierend auf Benutzereingaben.

Die Kombination dieser Funktionen eröffnet enorm viele Visualisierungsmöglichkeiten. Aus den JSON-Daten lassen sich Vektorgrafiken erstellen, die vom Browser gerendert werden können. Dadurch hat man bei der Visualisierung große gestalterische Freiheit. Man ist nicht auf die Darstellungsarten einer bestimmten Visualisierungsbibliothek beschränkt, sondern kann alles darstellen, was sich durch SVG und CSS beschreiben lässt.

Trotz dieser großen Flexibilität bleibt der Implementierungsaufwand vergleichsweise gering, da D3 diverse Layout-Algorithmen zur Verfügung stellt, die als Grundlage für gängige Darstellungsarten verwendet werden können. Ein in dieser Arbeit mehrfach verwendetes Layout, ist das sogenannte *Force-Layout*, ein kräftebasierter Algorithmus zum Zeichnen von Graphen. Er generiert ein visuell ansprechendes Layout, in dem versucht wird, Überlappung von Knoten und Kreuzung von Kanten zu vermeiden [49].

Verglichen mit Visualisierungsbibliotheken in anderen Programmiersprachen, sind die Interaktionsmöglichkeiten ein weiterer Vorteil von Javascript und D3. Javascript bietet ein umfassendes Event-System für Benutzerinteraktionen auf DOM-Elementen an und basierend darauf, lässt sich das DOM mit D3 einfach manipulieren. In Verbindung mit dem effizienten Rendering von Vektorgrafiken durch den Browser, lassen sich echtzeitfähige Darstellung mit beliebigen Interaktionsmöglichkeiten realisieren.

5.3.2 Interaktionskonzept

Wie in Kapitel 3.1.3 erläutert, spielt die Möglichkeit der Benutzerinteraktion bei der Exploration von Daten eine wichtige Rolle. Ausschlaggebend ist hier unter anderem die Wahl zwischen einem *Single-View*- und einem *Multi-View*-Ansatz. Beim Single-View-Ansatz werden alle Informationen in einer einzigen Darstellung vereint, während Mutli-View bedeutet, dass es mehrere verknüpfte Darstellungen gibt, die

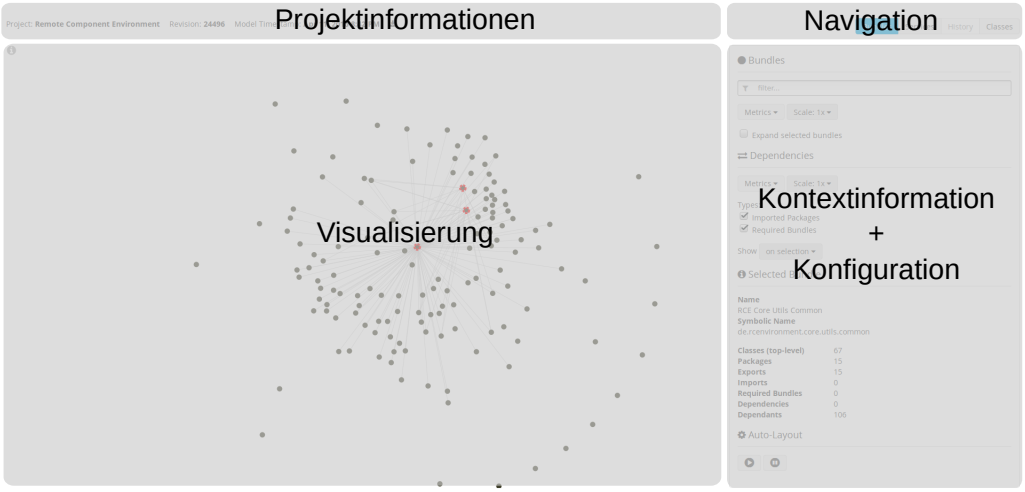


Abbildung 5.16: Aufteilung des GUI in vier Bereiche.

jeweils einen Teil der Informationen anzeigen. In der Softwarevisualisierung sind beide Ansätze vertreten [56, 36].

Durch die Auswahl mehrerer Darstellungsarten verfolgt diese Arbeit den Multi-View-Ansatz, auf dessen Vor- und Nachteile bereits in Abschnitt 3.2.2 eingegangen wurde. Demnach ist es wichtig, die Navigation zwischen den verschiedenen Views möglichst intuitiv zu gestalten und die kognitive Belastung des Nutzers gering zu halten. Deshalb sollte die Bedienung und der Aufbau des GUI view-übergreifend konsistent sein.

In Abbildung 5.16 ist der grundlegende Aufbau des implementierten GUIs skizziert. Am oberen Rand befindet sich eine Leiste mit statischen Informationen zum visualisierten Projekt. Rechts davon befinden sich Buttons zur Navigation zwischen den Views. Den Großteil des Bildschirms nimmt die eigentliche Visualisierung der aktuellen View ein und am rechten Rand befindet sich ein Panel, welches zum einen kontextabhängige Informationen und zum anderen Konfigurationsmöglichkeiten der aktuellen View anzeigt.

Nach dem Start der Visualisierung wird immer der Bundle-Graph angezeigt, der eine Übersicht über die komplette Software bietet. Ausgehend davon kann der Benutzer zu den drei anderen Views, Service-Graph, Klassen- und Package-Ansicht und Historienansicht, navigieren (Abbildung 5.17). Diese zeigen einen Ausschnitt der Software auf einer anderen Abstraktionsebene. Dieser Ausschnitt wird durch die Selektion eines oder mehrerer Bundles im Bundle-Graphen ausgewählt. Selektiert der Benutzer beispielsweise zwei Bundles im Bundle-Graph und wechselt danach zur Package- und Klassenansicht, sind dort nur die Packages und Klassen dieser beiden Bundles zu sehen.

Innerhalb der einzelnen Views stehen weitere, der in Abschnitt 3.1.3 genannten, Interaktionsmethoden, wie Filterung und Rekonfigurierung, zur Verfügung.

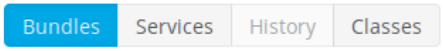


Abbildung 5.17: Navigationsleiste zum Wechsel zwischen den Views. Die Farben geben Hinweise, wo der Benutzer sich momentan befindent und welche Views von dort erreichbar sind.

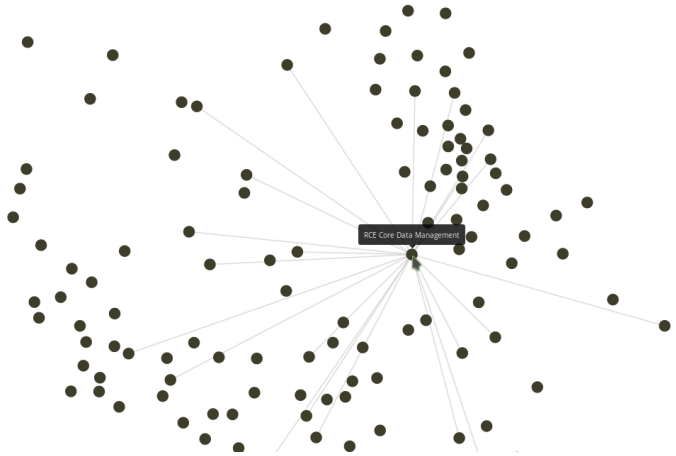


Abbildung 5.18: Bundle-Graph: Bei der Berührung eines Knoten mit der Maus, werden dessen adjazente Kanten angezeigt.

5.3.3 Bundle-Graph

Die zentrale Ansicht der Visualisierung ist der Bundle-Graph, der einen Überblick über alle Bundles der Software und deren Abhängigkeiten visualisiert.

Graph-Layout

Die Kanten des Graphen sind standardmäßig ausgeblendet und werden bei der Selektion eines Knotens oder beim Überfahren mit der Maus eingeblendet (Abbildung 5.18). Dies verringert den visuellen Overhead bei großen Graphen mit mehreren hundert Kanten.

Das Layout des Graphen entsteht durch den kräftebasierten Algorithmus (Force-Layout) von D3. Es ist dynamisch und wird in Echtzeit neu berechnet, wenn ein Knoten mit der Maus verschoben wird. Dieses Auto-Layouting kann jederzeit deaktiviert werden, so dass die Knoten frei angeordnet werden können und danach die festgelegte Position behalten. So kann der Benutzer bei Bedarf ein eigenes Layout erstellen, um bestimmte Aspekte des Graphen besser betrachten zu können.

Außerdem kann durch eine Zoom-Funktion der gezeigte Ausschnitt vergrößert und verschoben werden. Beim Bewegen der Maus über einen Knoten, wird ein Tooltip mit dem Namen des Bundles angezeigt.

Selektion

Um weitergehende Informationen zu einzelnen Bundles anzuzeigen, können diese durch Mausklick selektiert werden. Selektierte Knoten werden in der Darstellung

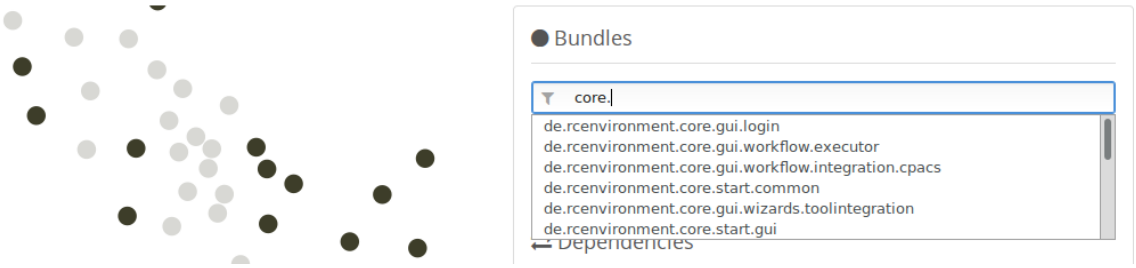


Abbildung 5.19: Filterung von Bundles über Suchfeld mit Autovervollständigung.

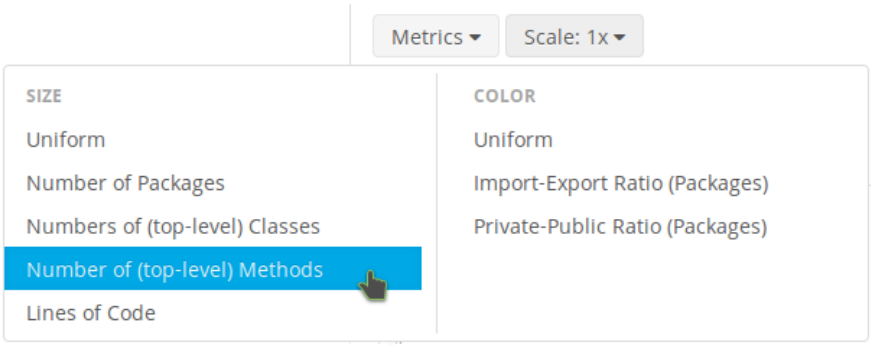


Abbildung 5.20: Dropdown-Menü zur Auswahl von Metriken.

mit einem roten Rand deutlich hervorgehoben. Das GUI-Panel am rechten Bildschirmrand zeigt Daten zum zuletzt selektierten Bundle an (siehe Abbildung 6.7). Neben der Bundle-Bezeichnung sind dies vor allem Zahlenwerte, wie die Anzahl von Klassen oder von ein- und ausgehenden Abhängigkeiten.

Filterung

Neben der Selektion ist die Filterung eine weitere zentrale Interaktionsmöglichkeit. Sie dient der Reduzierung der angezeigten Informationen und der Suche von Bundles, was besonders bei großen Graphen hilfreich ist.

Umgesetzt wurde die Filterung durch ein Suchfeld, in das Bundle-Namen oder Teile davon eingegeben werden können. In der Darstellung werden schon während des Tippens alle Knoten (und deren adjazente Kanten) ausgegraut, die nicht dem Suchbegriff entsprechen. Gleichzeitig klappt ein Dropdown-Menü mit Vorschlägen für die Autovervollständigung auf 5.19.

Metriken

Wie schon in Abschnitt 4.5 angedeutet, werden grafische Variablen der Darstellung dazu verwendet, um Metriken anzuzeigen. Da die Anzahl der möglichen Metriken sehr groß ist, hat der Benutzer die Möglichkeit, diese einzeln über Dropdown-Menüs auszuwählen (Abbildung 5.20).

Folgende grafische Variablen werden zur Visualisierung von Metriken benutzt:

1. Größe der Knoten

Abbildung 5.21: Graph mit visualisierten Metriken für die Größe von Bundles und derer Kopplung, die Richtung von Abhängigkeiten und das Verhältnis von Imports und Exports.

2. Farbe/Textur der Knoten

3. Dicke der Kanten

4. Farbe der Kanten

Die Größe der Knoten repräsentiert die Größe der Bundles. Hier sind Metriken, wie Anzahl an Klassen oder Lines-of-Code, wählbar. Durch anteilmäßiges Einfärben der Knoten werden Zahlenverhältnisse, wie zum Beispiel das Verhältnis von privaten und öffentlichen Klassen, dargestellt. Dadurch können, trotz des hohen Abstraktionsgrades[4] der Darstellung, Informationen über bundle-interne Strukturen vermittelt werden.

Die Dicke der Kanten eignet sich für die Darstellung der Stärke der Abhängigkeit (Kopplung). Als einfaches Maß für die Kopplung wurde die Anzahl importierter Packages zwischen den Bundles benutzt. Auch die Richtung der Abhängigkeiten sollte sichtbar sein. Da die klassische Art der gerichteten Kanten mit Pfeilspitzen bei großer Kantenzahl schnell unübersichtlich wird, wurde dafür stattdessen die Farbe der Kante verwendet. Sie zeigt einen Farbverlauf an, der zwischen zwei Farben an den Enden linear interpoliert.

Durch die Auswahl der Metriken kann der Benutzer sehr detailliert konfigurieren, was er in der Visualisierung sehen möchte und kann die Software aus verschiedensten Blickwinkeln betrachten.

Die implementierten Metriken sollen als Beispiele dienen. Auf Basis der verfügbaren Daten können selbstverständlich beliebige weitere Metriken aus Abschnitt 2.6 integriert werden. In Abbildung 5.21 ist ein Beispiel zu sehen, in dem die genannten Metriken angewandt wurden.

[4]Das heißt: niedriger *Level of Detail*

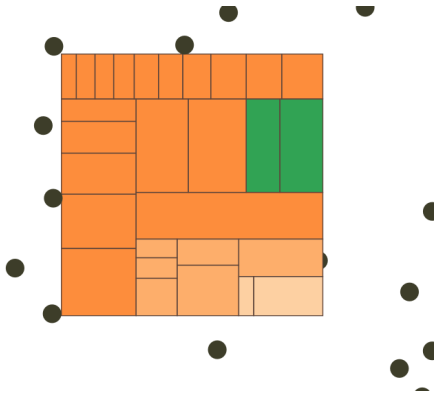


Abbildung 5.22: Bundle-Knoten, expandiert zu einer Treemap.

Detailansicht

Bei der Selektion eines Bundles kann optional eine Detailansicht des Bundles in Form einer Treemap eingeblendet werden. Dabei wird der kreisförmige Knoten durch ein deutlich größeres Quadrat ersetzt (Abbildung 5.22). Die Treemap stellt die Package- und Klassenstruktur dar, wobei jedes Rechteck eine Klasse repräsentiert und der Flächeninhalt von der Größe der Klasse abhängt. Alle Klassen eines Packages sind zudem in der gleichen Farbe eingefärbt.

Durch diese integrierte Detailansicht ist für die Betrachtung der bundle-internen Struktur keine separate View nötig und es können problemlos mehrere Bundles nebeneinander verglichen werden. Im Prinzip ähnelt dieser Ansatz einer sogenannten Fish-Eye-View, bei der eine Komponente detailliert und vergrößert dargestellt wird, die anderen Komponenten aber außen herum trotzdem noch (weniger detailliert) zu sehen sind. Der Vorteil ist, dass dadurch der Kontext sichtbar bleibt [20].

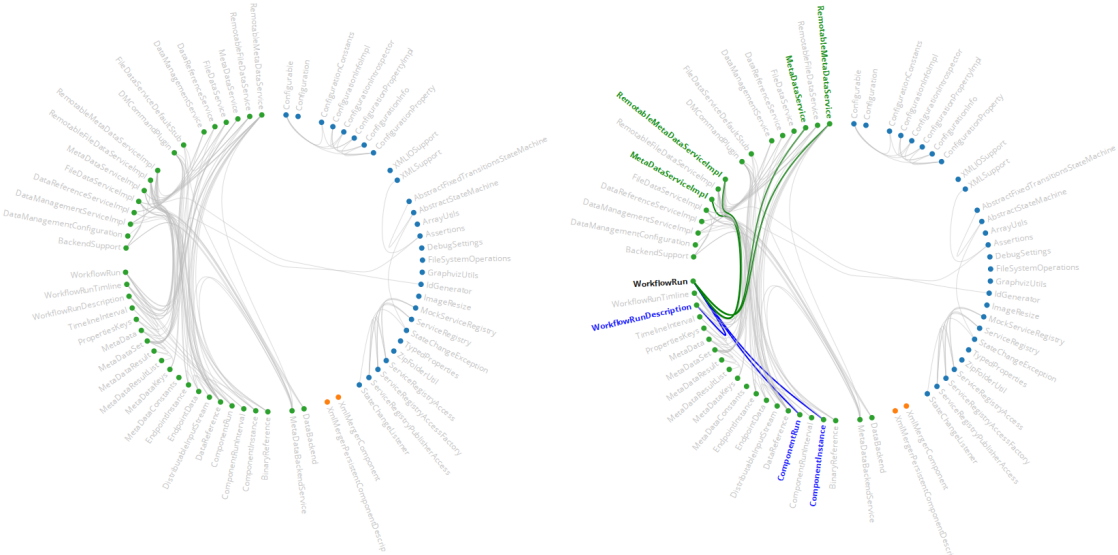
5.3.4 Klassen- und Package-Ansicht

Neben der integrierten Treemap im Bundle-Graph, die vor allem dazu gedacht ist, die Größe von Klassen und deren Packages zu visualisieren, gibt es eine zweite View auf der Granularität von Klassen, welche den Fokus auf deren Abhängigkeiten legt.

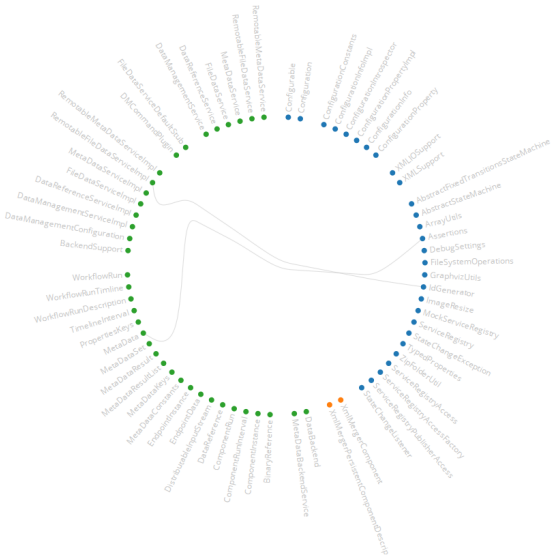
Abhängigkeiten zwischen Klassen

Die Klassen, von einem oder mehreren ausgewählten Bundles, werden als Kreise mit Beschriftung kreisförmig angeordnet (Abbildung 5.23a). Die Farbe der Knoten gibt die Zuordnung zu den Bundles an, wodurch ein Bezug zur vorherigen View (dem Bundle-Graph) hergestellt wird und die Klassen vom Benutzer besser eingeordnet werden können. Auch die Package-Aufteilung ist durch entsprechende Abstände zwischen den Knoten sichtbar.

Innerhalb des Kreises werden durch verbindende Linien Abhängigkeiten zwischen den Klassen visualisiert. Dabei kommt das, in Abschnitt 4.5 erwähnte Hierarchical-Edge-Bundling zum Einsatz, wodurch die Linien package-weise gebündelt werden.



(a) (b)



(c)

Abbildung 5.23: Darstellung der Abhängigkeiten auf Klassenebene mit Selektions- und Filtermöglichkeiten.

Abbildung 5.24: Graph aus Services (Kreise) und Service-Components (Quadrate). Die Farbe zeigt zu Bundle-Zugehörigkeit und die Form der Kante, ob es sich um das anbieten oder das referenzieren eines Services handelt.

Selektion und Filterung

Da auf Klassenebene zum Teil sehr viele Abhängigkeiten zustande kommen, können diese auch selektiv angezeigt und gefiltert werden. Um die Abhängigkeiten einer bestimmten Klasse hervorzuheben, genügt es, die Maus über die entsprechende Beschriftung zu bewegen. Dadurch werden ein- und ausgehende Abhängigkeiten (sowohl die Klassen, als auch die Verbindungen) farblich hervorgehoben. Über die Farbe wird zudem die Richtung der Abhängigkeit (ein- oder ausgehend) angezeigt (Abbildung 5.23b).

Um die Informationsdichte allgemein zu verringern, können verschiedene Filter angewandt werden. In Abbildung 5.23c zum Beispiel werden nur Abhängigkeiten eingeblendet, die über Bundle-Grenzen hinweg verlaufen.

Package-Hierarchie

Alternativ zu den Abhängigkeiten kann auch die Package-Hierarchie innerhalb des Kreises aus Klassen visualisiert werden. Dabei wird pro Bundle eine Baumstruktur mit Packages als innere Knoten eingeblendet. Somit weicht diese Darstellung etwas von der skizzierten Darstellung in Abbildung 4.1d ab. Gründe für diese Visualisierung von Packages sind vor allem technischer Natur.

5.3.5 Service-Graph

Die View, in der Services und Service-Components betrachtet werden können, ähnelt der Bundle-Ansicht. Auch hier werden Beziehungen in einem kräftebasierten Graph-Layout dargestellt (Abbildung 5.24).

Aufbau des Graphen

Ein Vorteil von serviceorientierter Architektur, ist die schwache Kopplung zwischen Komponenten. Service-Components stehen nicht in direkter Beziehung zueinander, sondern sind nur über Services miteinander verbunden (vgl. Metamodell in Abschnitt 5.1.2).

Dementsprechend gibt es zwei Arten von Knoten im Graph, die durch verschiedene Formen (Rechteck für Service-Components, Kreis für Services) repräsentiert werden. Ebenso können zwei Arten von Beziehungen vorkommen: das Anbieten und das Benutzen eines Services. Die entsprechenden Kanten werden anhand ihrer Form (gestrichelt und durchgezogen) sowie ihrer Farbe unterschieden.

Da die Beziehungen immer zwischen je einer Service-Component und einem Service bestehen und deren Ausrichtung stets von ersterer zur letzerer verläuft, ist eine explizite Darstellung der Kantenrichtung nicht nötig.

Interaktionsmöglichkeiten

Die Möglichkeiten zur Interaktion entsprechen etwa denen des oben vorgestellten Bundle-Graphs. Das Graph-Layout kann manuell angepasst werden und über ein Suchfeld können Informationen gefiltert werden.

5.3.6 Historienansicht

Die View zur Visualisierung von evolutionären Informationen wurde im Rahmen dieser Arbeit aus zeitlichen Gründen nicht umgesetzt. Da aber alle nötigen Daten von der Analyse bereitgestellt werden, kann dies problemlos nachgeholt werden.

Historische Informationen eignen sich gut für Metriken, die in die vorhandenen Views integriert werden könnten. Eine Idee wäre beispielsweise im Bundle-Graph mit der Farbe der Knoten Informationen, wie die Anzahl der Commits oder den Zeitraum seit dem letzten Commit, zu kodieren. Bei der Expansion eines Bundles könnten Bearbeitungsanteile von Autoren durch Flächen visualisiert werden oder es könnten Diagramme angezeigt werden, die beispielsweise Commits über die Zeit auftragen.

Trotz der möglichen Integration in andere Views ist es sinnvoll, den Graph mit historischen Informationen im GUI als separate View zu implementieren, so dass ein expliziter Wechsel dorthin nötig ist. So werden verschiedene Use-Cases besser getrennt und die Konfigurationsmöglichkeiten für den Benutzer können in den einzelnen Views übersichtlicher gehalten werden.

5.3.7 Implementierung

Die Visualisierung ist als eine einzige HTML-Seite implementiert. Diese lädt JavaScript-Code, der sich um die Darstellung und beim Wechsel der View um den Austausch von Seiteninhalt kümmert. Somit handelt es sich um eine *Single Page Application* (SPA), bei der zu keinem Zeitpunkt ein Neuladen der Seite durch

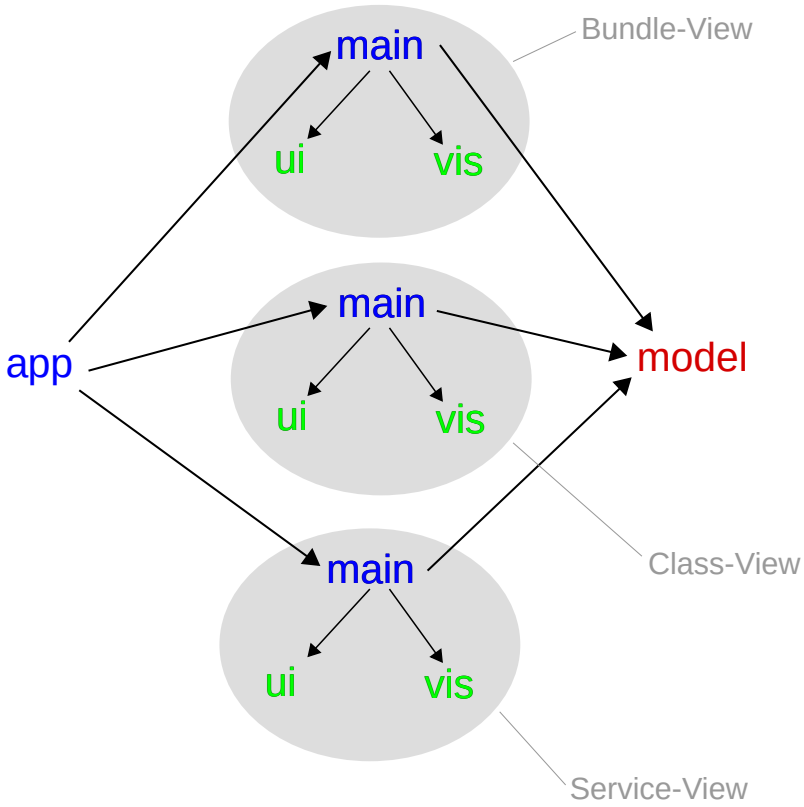


Abbildung 5.25: Interner Aufbau der Visualisierung. Die Module lassen sich in Model (rot), View (grün) und Controller (blau) einteilen. Jede View ist in sich gekapselt und eigenständig. Weitere Views können ergänzt werden.

den Browser nötig ist. Dadurch ähnelt die Benutzung der einer normalen Desktop-Anwendung [48]. Im Gegensatz zu den meisten anderen SPAs werden aber keine Daten von einem Server nachgeladen, da diese mit dem einmaligen Laden der JSON-Datei schon im Speicher liegen.

Architektur

Die Architektur der Anwendung ist an den *Model-View-Controller*-, bzw. den *Model-View-ViewModel*-Ansatz angelehnt. Das heißt, die Bereitstellung der Daten, die Generierung der Visualisierung und die Anwendungslogik, sind auf separate Module verteilt, die über bestimmte Schnittstellen kommunizieren. In Abbildung 5.25 ist die Modulstruktur leicht vereinfacht skizziert.

Zur Deklaration der Modulabhängigkeiten und zum Laden der Module zur Laufzeit, wird die Bibliothek *require.js* [100] benutzt. Sie implementiert den in JavaScript verbreiteten AMD-Standard (Asynchronous Module Definition) [85]. Damit können Abhängigkeiten effizient aufgelöst und asynchron geladen werden.

```
1
2  function bundleSizeNumberOfPackages(bundle) {
3    return model.getNumberOfPackageFragments(bundle);
4  }
5
6  var metrics = [
7    ...
8    { id: "bundles-size-metric-package-count",
9      name: "Number of Packages",
10     func: bundleSizeNumberOfPackages,
11     type: type.BUNDLE_SIZE
12   },
13   ...
14 ]
```

Listing 5.2: Definition einer Metrik durch eine Funktion und einen Array-Eintrag, der auf diese verweist.

Modelltransformationen

Auch in der Visualisierung ist eine Modelltransformation notwendig. Die Struktur der Daten ist darauf ausgelegt, für möglichst viele Visualisierungen nutzbar zu sein. Da die Layout-Algorithmen von D3 aber eigene Vorgaben machen, wie die Daten auszusehen haben, müssen sie vorher in das entsprechende Format transformiert werden. Um zum Beispiel die Treemap aus Abschnitt 5.3.3 berechnen zu können, benötigt der Layout-Algorithmus eine Baumstruktur mit Packages als innere Knoten und Klassen als Blätter, die so nicht direkt im Modell existiert.

Das in Abbildung 5.25 gezeigte *model*-Modul ist nicht nur dafür zuständig, die Daten aus der JSON-Datei zu laden, sondern führt auch die Modelltransformationen durch und stellt die Daten den einzelnen Visualisierungen zur Verfügung.

Erweiterbarkeit

Generell ist die Visualisierung so aufgebaut, dass eine Erweiterung jederzeit möglich ist. Wie in Abbildung 5.25 dargestellt, sind die einzelnen Views jeweils in eigene Module gekapselt, die relativ leicht ausgetauscht bzw. hinzugefügt werden können.

Die in den Views benutzten Metriken sind in einem sparaten Modul definiert und können ebenfalls leicht erweitert werden. Um eine neue Metrik hinzuzufügen, muss nur eine Funktion mit festgelegter Signatur definiert werden und die Metrik muss über einen Array-Eintrag deklarativ registriert werden. Daraufhin wird die Metrik automatisch in die View eingebunden und ein entsprechender Menüeintrag zur Auswahl der Metrik wird erstellt.

Beispielsweise ist die Metrik, welche die Größe eines Bundle-Knoten anhand der Anzahl an Packages im Bundle bestimmt, durch den in Listing 5.2 gezeigten Eintrag in `bundle/metrics.js` definiert.

6 Evaluierung

In diesem Kapitel wird die Evaluierung der Visualisierung vorgestellt, die am Beispiel einer Software vom Deutschen Zentrum für Luft- und Raumfahrt durchgeführt wurde. Tabelle 6.1 zeigt ausgewählte User-Stories, die typische Aufgaben beschreiben, für welche die Visualisierung eingesetzt werden könnte. Sie dienen als Grundlage für die Evaluierung.

6.1 Remote Component Environment

Die Software *Remote Component Environment* [87] (kurz RCE) bietet Wissenschaftlern und Ingenieuren die Möglichkeit, komplexe Berechnungs- und Simlationsworkflows zu modellieren und auszuführen. Ein *Workflow* besteht dabei aus verschiedenen Komponenten, wie zum Beispiel Simulationsprogammen, Werkzeugen zum Zugriff auf Daten oder benutzerdefinierten Skripten.

Diese Komponenten können vom Anwender in einer grafischen Oberfläche miteinander verbunden werden, wodurch ein Datenfluss zwischen ihnen definiert wird. In Abbildung 6.1 ist eine beispielhafte RCE-Instanz mit der Darstellung eines Workflows zu sehen. Neben vorgefertigten Komponenten, die häufig benötigte Funktionen zur Verfügung stellen, können Anwender auch selber Komponenten entwickeln, um eigene Tools in einen Workflow zu integrieren. Außerdem ist RCE eine verteilte Software, die es ermöglicht, Komponenten nicht nur lokal, sondern auch auf entfernten Systemen in anderen RCE-Instanzen auszuführen [65]. Somit ist RCE ein mächtiges Werkzeug für eine Vielzahl von Anwendungsfällen sowohl innerhalb als auch außerhalb des DLR.

RCE ist in Java implementiert und nutzt die *Eclipse Rich Client Platform* (RCP) [46]. Dabei handelt es sich um ein Framework zur Erstellung modularer Desktop-Anwendungen, die das Standard Widget Toolkit (SWT) als GUI-Bibliothek benutzen [53]. Eclipse RCP wiederum setzt auf OSGi auf, welches die Modularität und Flexibilität von RCP-Anwendungen erst möglich macht.

RCE ist ein Projekt, das seit 10 Jahren in Entwicklung ist und an dem momentan sechs Vollzeitmitarbeiter, sowie diverse studentische Hilfskräfte arbeiten. Es besteht aus etwa 2000 Java-Dateien in weit über 100 Modulen. Mit dem Ziel der Qualitätssicherung, ist das DLR an einer Visualisierung der Softwarearchitektur von RCE interessiert. Konkret soll diese beispielsweise mittelfristig dabei helfen, Abhängigkeiten zu entwirren und Schwachstellen im Design zu entdecken.

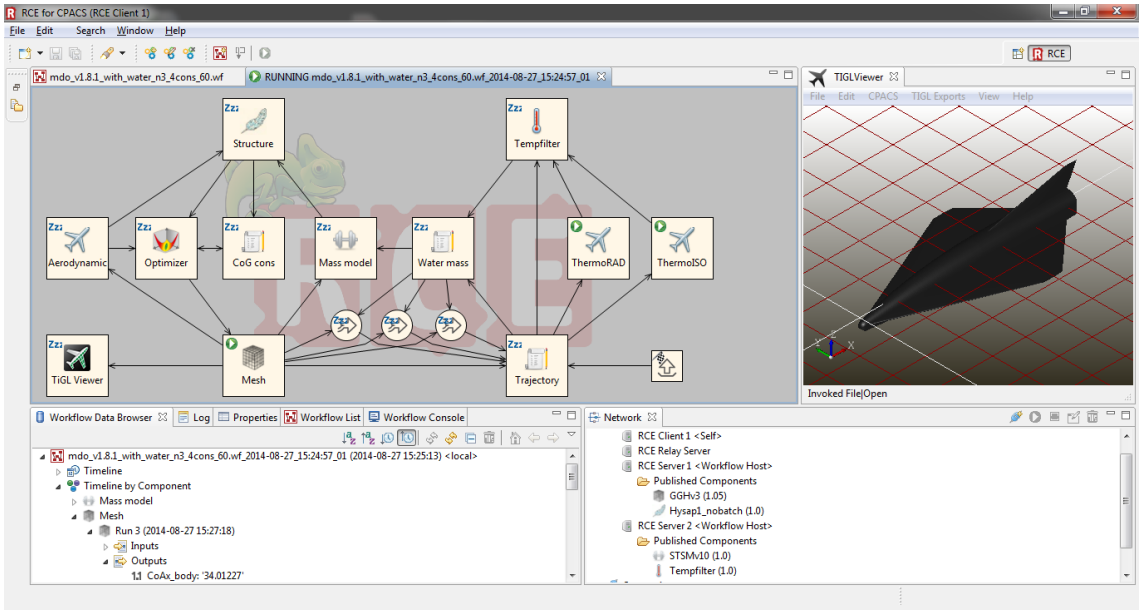


Abbildung 6.1: Screenshot einer RCE-Instanz, die den Workflow einer Optimierung am sogenannten SpaceLiner zeigt (Quelle: [87])

6.2 Konfiguration und Modellerstellung

Vor der eigentlichen Visualisierung musste das Datenmodell für RCE erstellt werden. Dabei wurden alle Test- und Feature-Bundles ausgeschlossen (siehe Anhang). Erstere enthalten zwar Quellcode, sind aber nicht Teil der Anwendung. Letzere enthalten keinen Code, sondern nur Metainformationen.

Zusätzlich mussten sieben weitere Bundles ausgeschlossen werden, die zur Laufzeit der Analyse Exceptions in der verwendeten Spoon-Bibliothek auslösten. Grund dafür sind Fehler in der Bibliothek, die bis zum Abschluss dieser Arbeit nicht behoben werden konnten.

Weitere Einschränkungen ergeben sich aus der Art, wie die Analyse abläuft. Da die Bundles einzeln analysiert werden und Abhängigkeiten zwischen Bundles auf Klassenebene nachträglich aufgelöst werden müssen (siehe Abschnitt 5.2.6), gibt es einige Fälle, in denen diese unvollständig sind. Beispiele dafür sind Verweise auf Klassen, die durch Wildcard-Import[1] importiert wurden oder, die anonym in einer anderen Klassen deklariert wurden. Referenzen auf solche Klassen sind im Modell nicht enthalten, was bei Betrachtung der Visualisierung im Hinterkopf behalten werden sollte.

Das Modell, welches von RCE erstellt wurde, umfasst letztendlich 142 Bundles. In serialisiertem Zustand beinhaltet die Modelldatei 306.264 Zeilen im JSON-Format und ist 11804 KiB groß. Die Generierung des Modells wurde auf einem Computer mit einem Intel i7-Prozessor aus dem Jahr 2009 mit 2,93GHz Grundtaktfrequenz ausgeführt und nahm im Schnitt zwischen 13 und 15 Minuten in Anspruch. Davon entfielen etwa 90% auf das Abrufen der Historie vom Subversion-Server. Grund dafür

[1]Zum Beispiel: `import com.example.*`

	Rolle	Ziel	Nutzen
6.3.1	Entwickler	Neuem Entwickler Überblick über Projekt geben	Einarbeitung neuer Mitarbeiter
6.3.2	”	Überblick und Einordnung eines unbekannten Moduls	Fähigkeit fremdes Modul weiter zu entwickeln
6.3.3	”	Review der Abhängigkeiten eines Moduls	Seiteneffekte einschätzen; Qualitätssicherung; Refactoring
6.3.4	Projektleiter	Ungerichtete Exploration zur Erkennung von Auffälligkeiten	Qualitätssicherung; Projektplanung
6.3.5	”	Überblick über Bereiche mit starken Veränderungen/Stagnation	”
6.3.5	”	Überblick darüber, wie das Wissen der Entwickler über den Code verteilt ist	”

Tabelle 6.1: User-Stories als Grundlage für die Evaluierung.

ist vermutlich, dass SVNKit, um Informationen über die 7000 analysierten Commits zu bekommen, ebenso viele Anfragen an den Server schicken muss.

6.3 User Stories

Im Folgenden wird die Visualisierung am Beispiel von RCE und für jede der, in Tabelle 6.1 gesammelten, User-Stories evaluiert[2]. Dazu werden beispielhaft Screenshots der Anwendung gezeigt, anhand derer erläutert wird, wo die Visualisierung einen Vorteil bringt und an welchen Stellen noch Verbesserungsbedarf besteht.

6.3.1 Überblick für neuen Entwickler

Im ersten Anwendungsfall wird davon ausgegangen, dass ein aktiver Entwickler des Projekts einem neuen Mitarbeiter mithilfe der Visualisierung einen Überblick über die Software geben möchte. Zunächst ist es sinnvoll, ein Gefühl für die Größe des Projekts zu vermitteln. In Abbildung 6.2 ist die Visualisierung zu sehen, wie sie unmittelbar nach dem Start aussieht. Diese Ansicht vermittelt schnell einen ersten Eindruck von der Anzahl der Module. Konkrete Zahlen zur Projektgröße sind im Dropdown-Menü am oberen Bildschirmrand zu finden.

Beim Einstieg in eine neue Software ist es wichtig, deren zentrale Module kennenzulernen. Auch hier ist die Größe ein guter Anhaltspunkt. Durch Einsatz der LOC-Metrik kann der Umfang der einzelnen Bundles betrachtet werden (Abbildung 6.3a).

[2]Die Nummern in der Tabelle verweisen auf die zugehörigen Abschnitte.

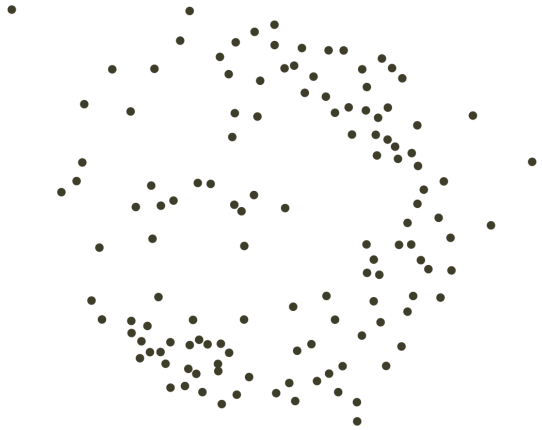
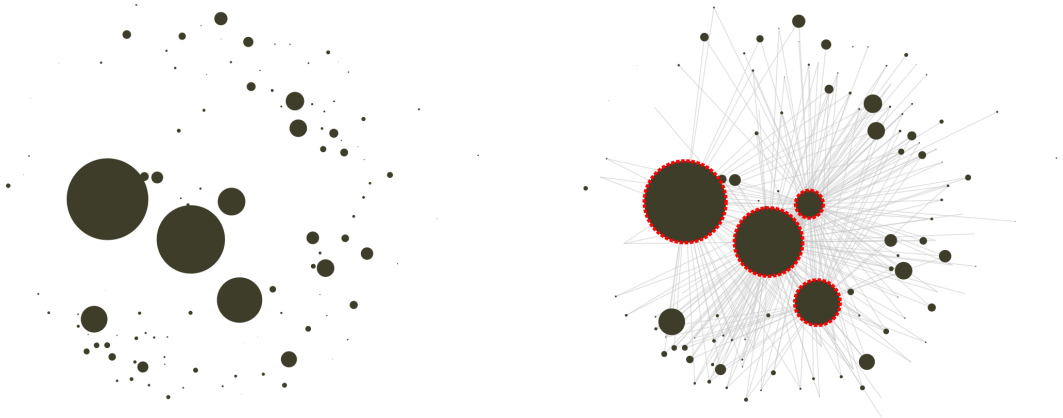
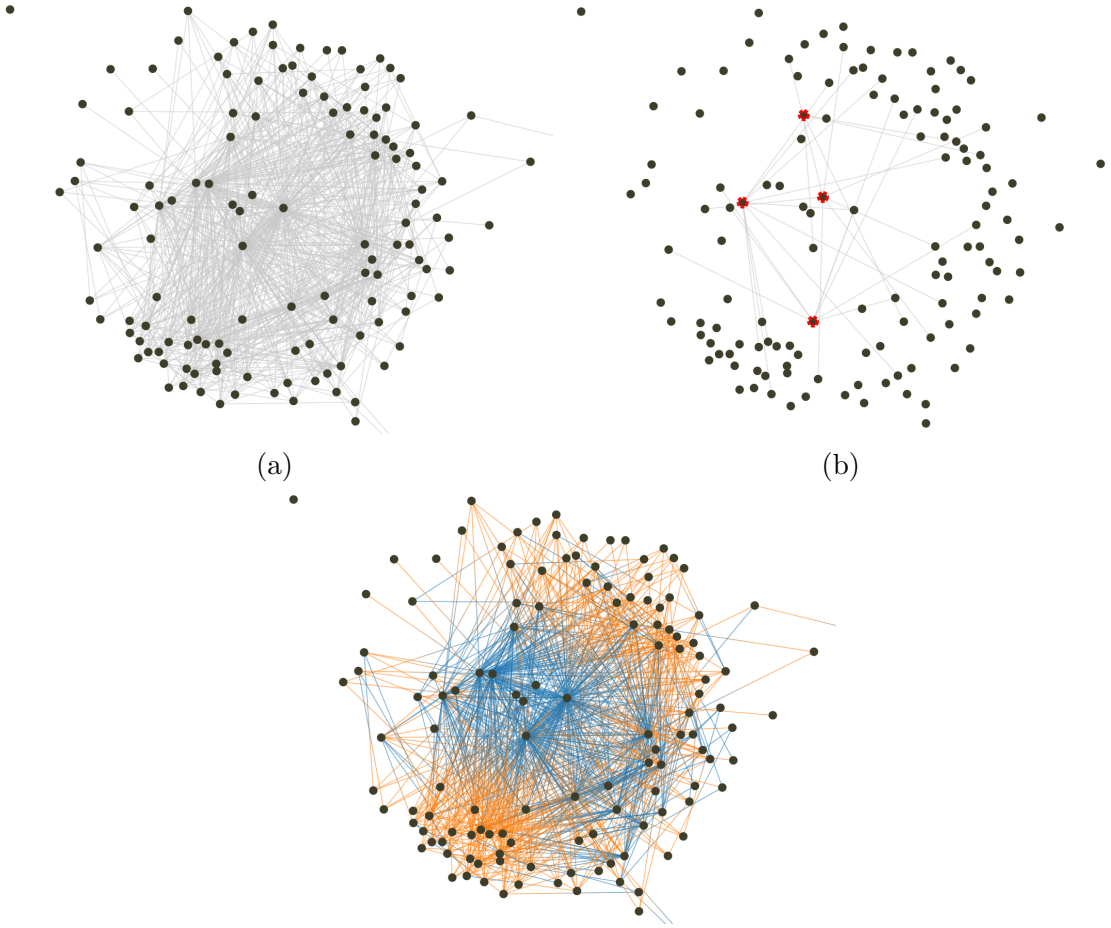


Abbildung 6.2: Punktwolke der Bundles von RCE.



(a) (b)

Abbildung 6.3: Bundles mit Größenverhältnissen nach der LOC-Metrik (a) und zusätzlich eingeblendeten Beziehungen der vier größten Bundles (b).



(a) (b)

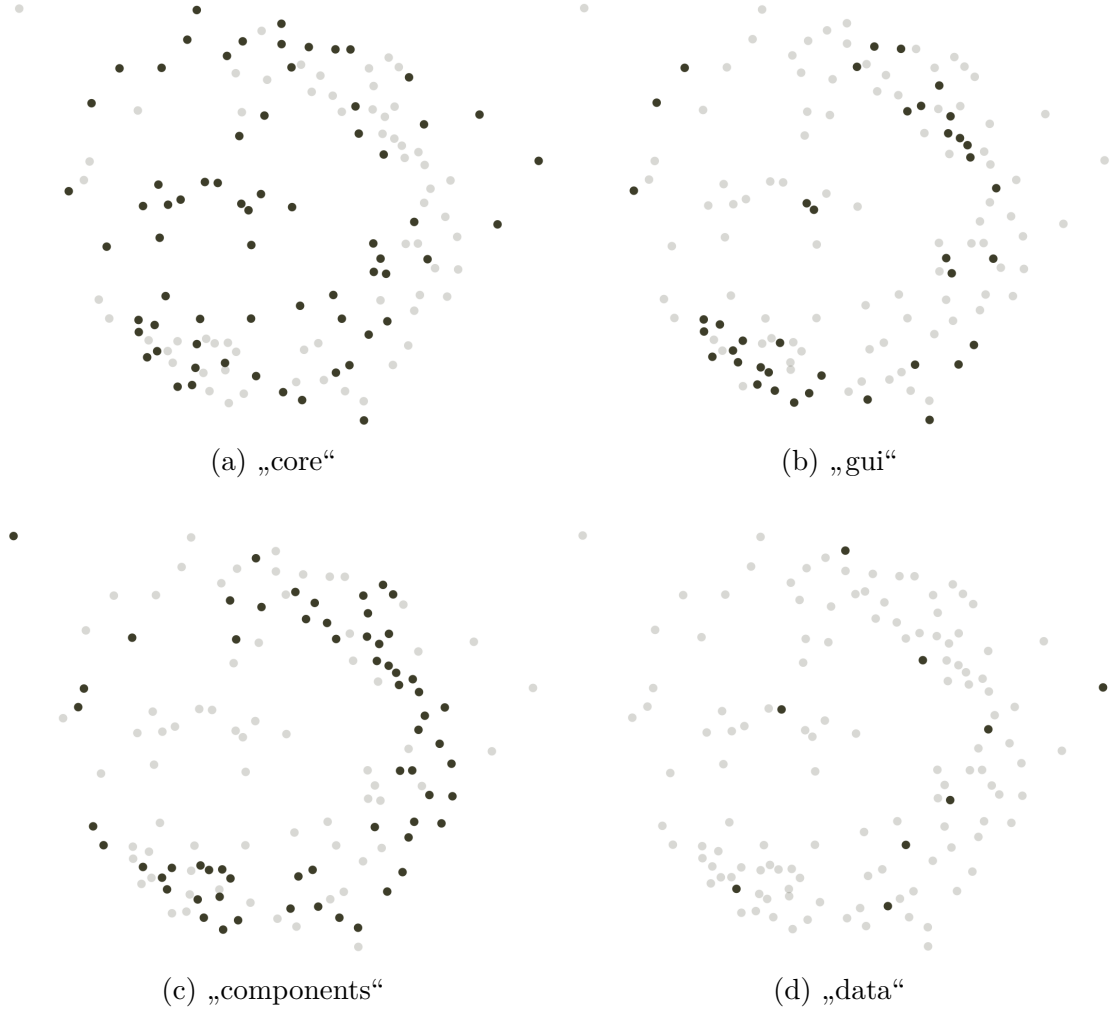
(c)

Abbildung 6.4: Abhängigkeiten zwischen Bundles.

Die größten Bundles fallen dabei direkt ins Auge und durch Überfahren der Knoten mit der Maus lässt sich feststellen, dass die vier größten Bundles in RCE *Core Communication*, *Core Component*, *Core Workflow* und *Core Utils Common* sind. Die Bezeichnungen legen, auch für eine projektfremde Person, nahe, dass es sich tatsächlich um zentrale Komponenten handelt. Das Einblenden der Abhängigkeiten bestätigt diesen Eindruck (Abbildung 6.3b).

Auch das Layout des Graphen kann Anhaltspunkte für wichtige Bundles liefern (Abbildung 6.4a). Allerdings nur, wenn Bundles aufgrund ihrer großen Anzahl von Beziehungen als *wichtig* bezeichnet werden. Durch das kräftebasierte Layout werden Bundles mit vielen Verbindungen zu anderen Bundles tendenziell eher in der Mitte der Punktwolke platziert, während sich Bundles mit wenigen Verbindungen am äußeren Rand befinden. Wie in Abbildung 6.4b zu sehen ist, kann es aber auch Ausnahmen geben, weshalb dieser Annahme nicht zu viel Gewicht verliehen werden sollte.

Dass zentrale Bundles häufig benutzt werden, also viele eingehende Abhängigkeiten haben, ist gut zu sehen, wenn man die Richtung der Beziehungen betrachtet. Besonders bei einer großen Anzahl an Kanten, ist die farbliche Darstellung sehr



(a) „core“ (b) „gui“

(c) „components“ (d) „data“

Abbildung 6.5: Bundles gefiltert nach verschiedenen Strings.

prägnant (Abbildung 6.4c).

Generell fällt bei der Benutzung auf, dass es möglicherweise hilfreich sein könnte, wenn die Bundle-Namen dauerhaft, also nicht nur bei Mausberührung, angezeigt werden würden, um diese Informationen auf einen Blick bekommen zu können. Ob dies tatsächlich praktikabel ist oder durch die große Anzahl an Komponenten zu unübersichtlich wäre, müsste getestet werden.

Neben dem Hinweis auf besonders wichtige Bundles, ist es für einen neuen Entwickler auch gut zu wissen, wie er einzelne Bundles bestimmten Funktionalitäten oder Bereichen der Software zuordnen kann. In großen Projekten gibt es meist Module, die noch über der Ebene von Bundles definiert sind. So lassen sich Bundles in RCE durch ihre (hierarchischen) Bezeichnungen gruppieren. Da diese Bezeichnungen sehr projektspezifisch und nicht einheitlich sind, lässt sich diese Gruppierung schwer automatisiert analysieren.

Die interaktive Filterfunktion kann dazu genutzt werden, einen neuen Entwickler auf bestimmte Bundle-Gruppen aufmerksam zu machen. In Abbildung 6.5 sind Bundles hervorgehoben die durch die Strings „core“, „gui“, „components“ und „da-

ta“ gefiltert wurden.

Erwartungsgemäß sind „components“-[3] und „gui“-Bundles eher am Rand der Darstellung zu finden. Ausnahmen sind die Bundles *Core GUI Resources* und *Core GUI Utils*.

Neben der Bundle-Ansicht ist auch die Betrachtung von Services für einen neuen Entwickler relevant, da es sich dabei um ein zentrales Konzept von OSGi handelt. In Abbildung 6.6 ist der komplette Service-Graph von RCE zu sehen. Dieser eignet sich offensichtlich nicht, um einen Überblick über die Services im Gesamten zu erhalten. Wegen der großen Anzahl an Knoten ist das Layout sehr unübersichtlich und die Farben helfen bei der Orientierung nicht, da es zu viele gibt, um sie auf einen Blick unterscheiden zu können. Trotzdem stechen einige Services visuell dadurch hervor, dass sie besonders viele verbundene Service-Components haben, was zu einer kreisförmigen Anordnung führt. An dieser Stelle kommt, die in Abschnitt 4.2 erwähnte Fähigkeit des menschlichen Wahrnehmungssystems, zu tragen, gut Muster erkennen zu können.

Insgesamt lässt sich aber sagen, dass der Service-Graph, wenn überhaupt, nur für die Betrachtung ausgewählter Services geeignet ist. Das heißt, vor dem Wechsel in die Service-View, sollte eine begrenzte Anzahl an Bundles selektiert werden.

6.3.2 Einordnung eines fremden Moduls

Bekommt ein Entwickler die Aufgabe, an einem für ihn unbekannten Modul zu arbeiten, muss er zunächst einen Überblick über das Modul bekommen und es in den Kontext der Anwendung einordnen, um an vorhandenes Wissen anknüpfen zu können. Über die Filterfunktion kann das Bundle schnell gefunden werden. Die Metriken für Modulgrößen können zum Vergleich von Bundles benutzt werden. Nach der Selektion des Bundles können Detailinformationen im GUI abgelesen werden und Metriken, wie das Verhältnis von internem und öffentlichen Code, helfen dabei, das Bundle grob zu klassifizieren (Abbildung 6.7).

Um einen ersten Eindruck über die innere Struktur des Bundles zu bekommen, kann der Knoten zur Treemap expandiert werden. Im Beispiel aus Abbildung 6.8 ist zu sehen, dass es fünf Packages gibt. Das grün gefärbte Package enthält zwar nur zwei Klassen, wovon eine aber zu den größten des Bundles gehört. An dieser Stelle wäre es hilfreich, weitere Informationen über die Klasse bekommen zu können, wie etwa die genaue LOC-Zahl oder die Anzahl an Methoden. Die nötigen Daten sind vorhanden und könnten leicht in die Darstellung integriert werden.

Desweiteren wäre es denkbar, die internen und/oder externen Abhängigkeiten in der Treemap einzublenden. Andererseits ist dafür die separate View auf Klassenebene vorgesehen, in die bei Bedarf gewechselt werden kann.

Für Abbildung 6.9a wurde dafür ein weiteres verbundenes Bundle selektiert, um nicht nur interne, sondern auch ausgewählte externe Abhängigkeiten zu sehen. Dort kann man beispielsweise die oben erwähnte, besonders große Klasse selektieren und

[3] Als *Component* werden in RCE Bundles bezeichnet, die nicht zum Kern (*core*) der Software gehören. Components können die API vom Kern benutzen. Eine Abhängigkeit in die entgegengesetzte Richtung kann es aber nicht geben.

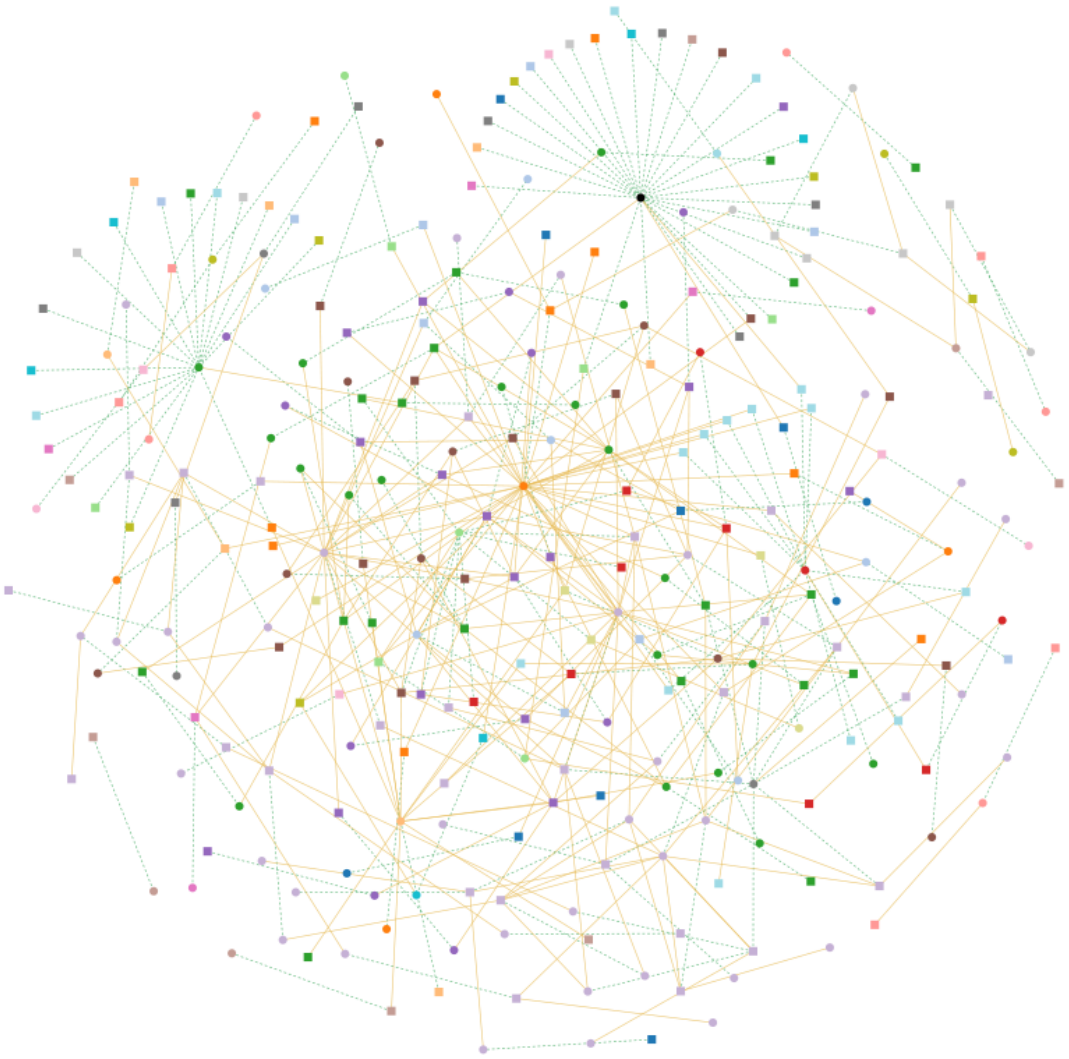


Abbildung 6.6: Übersicht über alle Services und Service-Components von RCE.

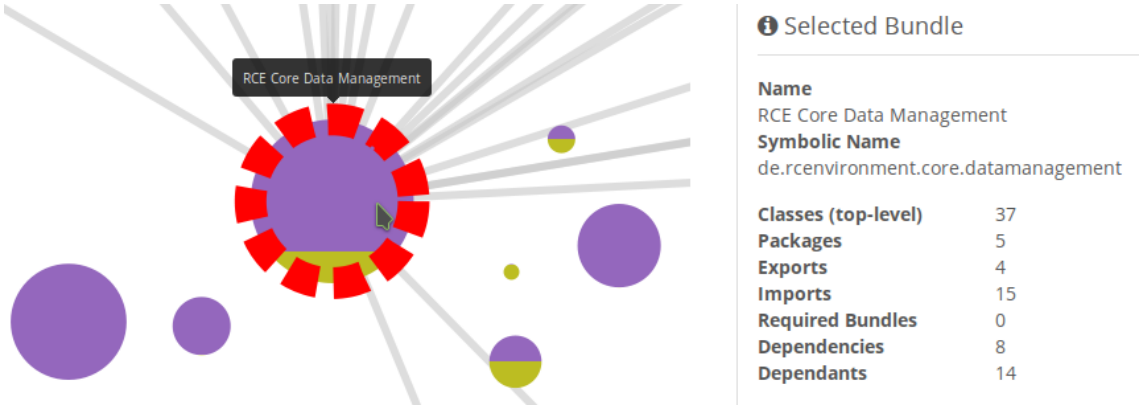


Abbildung 6.7: Informationen zu einem selektierten Bundle.

Abbildung 6.8: Treemap-Darstellung eines Bundles.

feststellen, dass sie sowohl vergleichsweise viele externe, als auch interne Abhängigkeiten zu anderen Klassen besitzt (Abbildung 6.9b). Anhand der blauen Färbung lässt sich gleichzeitig ablesen, dass es sich dabei nur um ausgehende Abhängigkeiten handelt, die Klasse also keine direkten Seiteneffekte auf anderen Code hat.

Durch die verschiedenen Ansichten von Bundles hat der Entwickler diverse Möglichkeiten, dieses auf verschiedenen Abstraktionsebenen kennenzulernen und einen Überblick über Struktur und Abhängigkeiten zu bekommen.

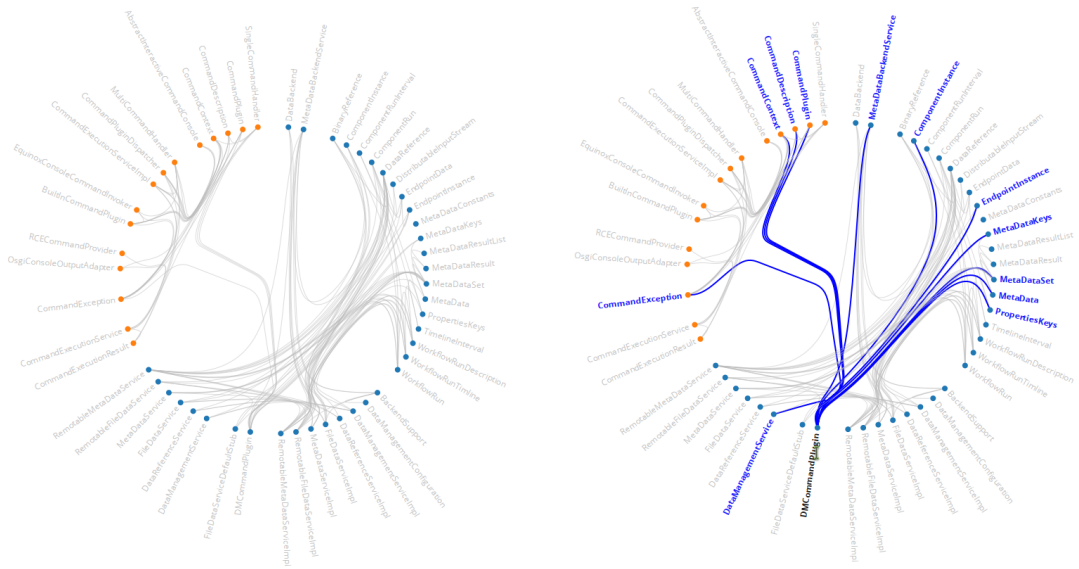
6.3.3 Review von Abhängigkeiten

Der Fokus in dieser Arbeit liegt auf Beziehungen, vor allem Abhängigkeiten. Deshalb sollte es einem Entwickler möglich sein, durch die Visualisierung Abhängigkeiten von einem oder mehreren Bundles zu untersuchen.

Betrachtet man die Abhängigkeiten im Bundle-Graph von RCE, ist nicht nur auffällig, dass, wie schon erwähnt, die zentralen Module größtenteils eingehende Abhängigkeiten haben. Es fallen auch besonders starke Abhängigkeiten ins Auge (Abbildung 6.10a). Möchte man eine dieser Abhängigkeiten genauer analysieren, kann sie durch Selektion und Filterung, wie in Abbildung 6.10b, freigestellt werden. Dadurch sieht man, dass im Beispiel ein kleines Bundle scheinbar sehr viele Packages eines deutlich größeren Bundles importiert.

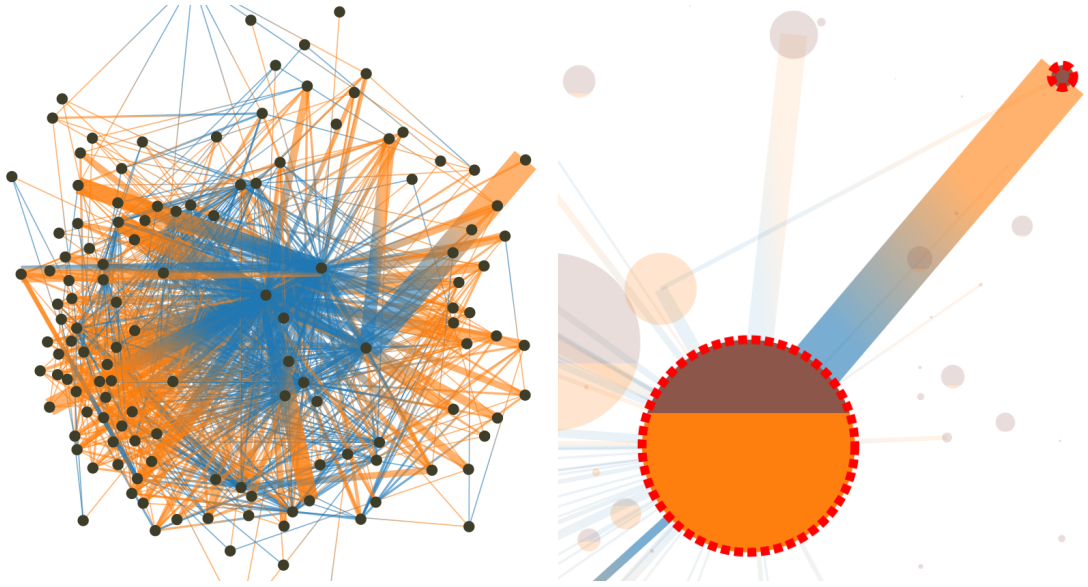
Beim Wechsel in die Klassenansicht fällt jedoch auf, dass das kleinere Bundle (blau) tatsächlich nur Klassen aus 9 von 25 exportierten Packages des größeren Bundles (orange) referenziert (Abbildung 6.11). Diese Feststellung ist erst durch die Filterungsmöglichkeiten sowie die visuelle Gruppierung der Klassen nach Packages und Bundles möglich.

Der scheinbare Widerspruch zwischen wirklich benutzten Klassen und Stärke der Abhängigkeit lässt sich durch erneuten Blick in den Bundle-Graph erklären. Blendet man dort alle Abhängigkeiten aus, die durch eine `Import-Package`-Anweisung zustande kommen, bleibt die Verbindung zwischen den beiden Bundles sichtbar. Es handelt sich also um eine Abhängigkeit durch eine `Require-Bundle`-Anweisung, was



(a) (b)

Abbildung 6.9: Abhängigkeiten auf Klassenebene



(a) (b)

Abbildung 6.10: Kopplung zwischen Bundles.

(a) (b)

Abbildung 6.11: Referenzen zwischen Klassen zweier Bundles: vollständig (a) und nur bundle-übergreifend (b).

dazu führt, dass implizit alle Packages des größeren Bundles importiert werden.

Dieses Beispiel zeigt deutilch, wie die Kombination der verschiedenen Views genutzt werden kann, um die Abhängigkeiten des Systems gezielt zu analysieren und so mögliche Schwächen im Design aufzudecken.

6.3.4 Erkennung von Auffälligkeiten

Durch die Visualisierung soll die analysierte Software grafisch und interaktiv erforschbar werden. Ziel einer solchen Exploration könnte das Finden von Auffälligkeiten sein, die auf Fehler oder Designmängel hindeuten. Im vorherigen Abschnitt wurde schon ein Beispiel vorgestellt, in dem so eine unnötig starke Abhängigkeit zwischen zwei Bundles gefunden wurde.

Bei der Evaluierung wurden weitere Stellen in RCE gefunden, die zwar nicht unbedingt fehlerhaft sind, aber durch bestimmte Eigenschaften auffallen.

Die Treemap in Abbildung 6.11b zum Beispiel, zeigt ein vergleichsweise großes Bundle, in dem es eine Klasse gibt, deren Größe deutlich hervorsticht. Hier ist es schwierig, weitere Aussagen mit der Visualisierung zu treffen, da es keine Ansicht gibt, die das Innere von Klassen darstellt. Eine direkte Verknüpfung zum Quellcode, um diesen bei Bedarf anzuzeigen, gibt es ebenfalls nicht. Dies wäre eine sinnvolle Erweiterung.

Im Bundle-Graph sind mehrere Knoten zu sehen, die sich am Rand der Darstellung befinden und keinerlei Verbindung zu anderen Bundles aufweisen (Abbildung 6.12). Dafür kann es verschiedenste Gründe geben. Das Bundle könnte im Laufe der Zeit überflüssig geworden sein (*Dead Code*). Es könnte aber auch über ein Bundle mit dem Rest der Codebasis verknüpft sein, das nicht in die Analyse einbezogen wurde.

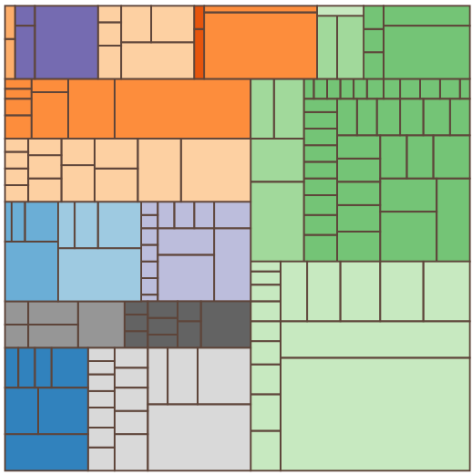


Abbildung 6.12: Treemap-Darstellung mit auffällig großer Klasse (unten rechts).

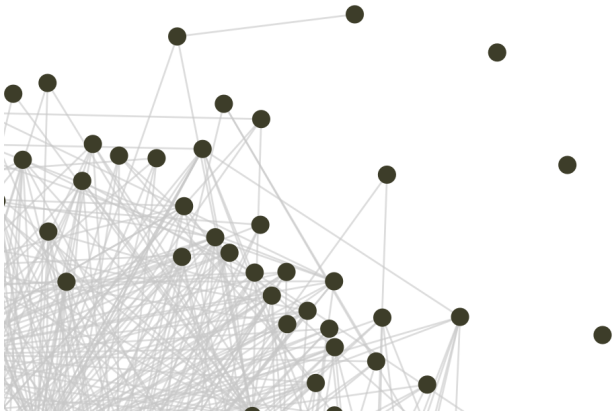


Abbildung 6.13: Bundles ohne Verbindung zur restlichen Codebasis.

Dies ist nicht unwahrscheinlich, da wie schon erwähnt, mehrere Bundles in der Konfiguration ausgeschlossen wurden. Desweiteren gibt es durch den Aufbau des Eclipse RCP Frameworks die Möglichkeit, dass Bundles von diesem geladen werden, ohne direkten Bezug zu den anderen Bundles der Anwendung zu haben.

Auch harmlose Verstöße gegen Projekt-Guidelines können über die Visualisierung erkannt werden. So fiel beispielsweise ein Bundle auf, welches trotz Einsatz einer entsprechenden Metrik, farblos dargestellt wurde, was zeigt, dass es keinen Quellcode enthält. Grund dafür war allerdings, dass der enthaltene Code nur einem abweichenden Verzeichnis innerhalb des Bundles lag und so nicht von der Analyse gefunden werden konnte.

An diesen Beispielen wird noch einmal deutlich, worauf bereits in Abschnitt 4.2 eingegangen wurde: Die Visualisierung kann immer nur *Hinweise* auf Auffälligkeiten geben, es bleibt aber die Aufgabe eines Entwicklers, zu entscheiden, ob etwa ein Refactoring nötig ist.

6.3.5 Untersuchung von Entwicklungsaktivität und Wissensverteilung

Die letzten beiden User-Stories können nicht anhand der Visualisierung evaluiert werden, da diese noch keine Darstellung für evolutionäre Daten aus dem Versionsverwaltungssystem beinhaltet. Im Datenmodell sind die nötigen Daten aber schon vorhanden. Aus diesen wurden beispielhaft zwei Statistiken erstellt, die in den Tabellen 6.2 und 6.3 zu sehen sind.

Erstere zeigt wann, wie viele Bundles zuletzt geändert wurden und ist ein Beispiel für die Analyse der Entwicklungsaktivität. Auffällig ist hierbei, dass kein Bundle älter als acht Wochen ist. Augenscheinlich fanden vor sieben bis acht Wochen großflächige Modifikationen statt. Hier wird die Schwierigkeit bei der Interpretation von historischen Daten deutlich: Ob es sich um fundamentale und komplexe oder eher triviale Änderungen (wie zum Beispiel Anpassung von Copyright-Headern) handelt, kann ohne detailliertere, semantische Analyse nicht bestimmt werden.

Tabelle 6.3 zeigt wie die Daten genutzt werden können, um Hinweise auf die Wissensverteilung der Entwickler im Code zu bekommen. In diesem Beispiel wurden die Commits von 17 Autoren in 129 Bundles analysiert. Wie zu sehen ist, gibt es 21 Bundles, die von jeweils nur einem Autor entwickelt wurden. An einem Großteil der Bundles sind aber zwei bis fünf Autoren beteiligt.

Letzte Änderung (Wochen)	< 1	1-2	2-3	3-4	4-5	5-6	6-7	7-8	$8 <$
Anzahl Bundles	17	9	9	0	22	6	0	66	0

Tabelle 6.2: Alter der letzten Änderung in Wochen und Anzahl an Bundles für die dies zutrifft.

Anzahl Autoren	1	2	3	4	5	6	7	8	9	10	11
Anzahl Bundles	21	19	25	24	19	9	6	3	1	1	1

Tabelle 6.3: Anzahl beteiligter Autoren und Anzahl an Bundles für die dies zutrifft.

Diese Beispiele zeigen, dass das Datenmodell die nötigen Informationen zur Analyse der Softwarehistorie bereitstellt. Eine Integration dieser Daten in die Visualisierung steht aber noch aus.

6.4 Fazit

Als Ergebnis der Evaluierung lässt sich festhalten, dass die Visualisierung in verschiedenen Anwendungsfällen von Nutzen ist. Sie eignet sich gut, um einen Überblick über die Softwarearchitektur zu bekommen und sich mit fremden Modulen und deren Abhängigkeiten vertraut zu machen. Der Abhängigkeitsgraph der Bundles, mit

einblendbarer Treemap und das kreisförmige Netz zur Darstellung der Beziehungen auf Klassenebene bieten viele Möglichkeiten, die Software aus verschiedenen Blickwinkeln zu erkunden. Der Einsatz von Metriken und der Filterfunktion leistet einen entscheidenden Beitrag zum Erkenntnisgewinn.

Der Nutzen der Serviceansicht hingegen ist noch stark begrenzt. Das kräftebasierte Graph-Layout scheint an dieser Stelle ungeeignet zu sein. Nicht nur bei großen Graphen, sondern auch bei kleiner Anzahl an Knoten, macht es einen unübersichtlichen Eindruck. Dies hängt vermutlich mit der speziellen Struktur von Graphen aus Service und Service-Components zusammen.

Nicht nur zum Kennenlernen von fremden Code, sondern auch zur Suche von Auffälligkeiten in bekanntem Code, ist die Visualisierung gut geeignet. Die beschriebenen Beispiele haben gezeigt, dass hier besonders die Kombination mehrerer Views auf unterschiedlichen Abstraktionsebenen sinnvoll ist. Trotzdem wäre eine weitere View auf Klassen- und Methodenebene sowie eine direkte Verknüpfung zum Quellcode hilfreich. Dies würde die bestehende Lücke zwischen Visualisierung und Quelltext schließen.

Eine View zur Visualisierung der Software-Evolution steht noch aus. Wie gezeigt wurde, stehen dafür aber alle nötigen Daten zur Verfügung.

7 Abschluss

In diesem Kapitel werden Inhalte und Ergebnisse dieser Arbeit zusammengefasst. Anschließend wird ein Ausblick auf mögliche Verbesserungen und Erweiterungen des vorgestellten Ansatzes gegeben.

7.1 Zusammenfassung

Softwarevisualisierung kann dazu beitragen, die Architektur von Software zu vereinfachen und zu verstehen. Da es für OSGi-Software bisher keine umfassenden und praxistauglichen Visualisierungsanwendungen gibt, wurde eine solche konzipiert und beispielhaft implementiert. Dazu wurde zunächst analysiert, welche Aspekte von OSGi-Software von Interesse sind. Der Fokus lag dabei auf den Abhängigkeiten zwischen Komponenten wie Bundles, Services und Klassen.

Die ausgewählten Aspekte wurde in einem Metamodell festgehalten. Anschließend wurde eine Java-Anwendung erstellt, die die benötigten Daten aus einem zu analysierenden Softwareprojekt extrahiert und in ein Modell transformiert.

Aufbauend auf den theoretischen Grundlagen der Datenvisualisierung und bisherigen Ansätzen der Softwarevisualisierung wurde eine Visualisierung mit mehreren Views konzipiert. Diese benutzen verschiedene Darstellungsarten wie Graphen und Treemaps, um das Modell der analysierten Software grafisch auf unterschiedlichen Abstraktionsebenen darzustellen. Dabei kamen geeignete Software-Metriken zum Einsatz, die auf visuelle Parameter abgebildet wurden und so einen detaillierten Einblick in die Software bieten. Interaktionsmethoden wie Navigation, Selektion und Filterung, ermöglichen dem Betrachter eine Exploration der Architektur.

Am Beispiel einer OSGi-Software vom Deutschen Zentrum für Luft- und Raumfahrt wurde die Visualisierung, hinsichtlich verschiedener User-Stories, evaluiert. Wie in Abschnitt 6.4 erläutert, zeigt die Evaluierung, dass die umgesetzt Visualisierung einen effektiven Nutzen in unterschiedlichen Anwendungsfällen bieten kann. Trotzdem gibt es Potential für Verbesserungen und Erweiterungen.

7.2 Ausblick

In Kapitel 5, das die Umsetzung des vorgestellten Konzepts erläutert, wurde schon auf Erweiterungsmöglichkeiten des Metamodells (Abschnitt 5.1.3) und der Analyse (Abschnitt 5.2.7) eingegangen. In Abschnitt 6.4 der Evaluierung wurde beschrieben, wo es Verbesserungsbedarf in der Visualisierung gibt.

An dieser Stelle sollen deshalb nur einige besonders viel versprechende Punkte zusammengefasst werden, an denen bei einer Weiterentwicklung von Konzept und

Implementierung angesetzt werden könnte.

Der Ansatz dieser Arbeit konzentriert sich auf die Kopplung und einige strukturelle Aspekte wie Größen und Hierarchien von Softwarekomponenten. In Abschnitt 2.6.3 wurden aber auch Metriken für die Kohäsion von Modulen vorgestellt. Diese könnten ebenfalls in die Visualisierung integiert werden, wobei dafür das Metamodell stellenweise erweitert werden müsste. Überhaupt nicht betrachtet, aber möglicherweise auch lohnenswert, wären Metriken speziell für serviceorientierte Architekturen [17, 59, 67]. Wie in Abschnitt 5.3.7 beschrieben, ist die Visualisierung so entworfen, dass ein Hinzufügen weiterer Metriken wenig Implementierungsaufwand bedeutet.

Auch ohne eine Erweiterung des vorhandenen Datenmodells bietet es Daten, die in der Visualisierung noch nicht berücksichtigt werden. Neben den detaillierten Informationen zu einzelnen Klassen, ist das vor allem die Historie der Software. Ideen für eine ensprechende View wurden in Abschnitt 5.3.6 beschrieben. Deren Umsetzung würde die Einsatzzwecke der Visualisierung deutlich erweitern.

Wie schon erwähnt, muss besonders der Graph aus Services und Service-Components (Abschnitt 5.3.5) verbessert werden, um einen praktischen Nutzen zu bringen. Da auch die Anpassung von Parametern das kräftebasierte Layout des Graphen nicht verbessern konnte, wäre hier zu überlegen, ob ein anderer Algorithmus zum Zeichnen von Graphen Vorteile bringt. Zudem könnte evaluiert werden, ob das Clustern der Knoten nach Bundle-Zugehörigkeit eine bessere Interpretation der Darstellung ermöglicht.

Auch in den anderen Views gibt es Verbesserungspotential. Die kreisförmige Anordnung von Klassen (Abschnitt 5.3.4) beispielsweise, bietet zwar einen kompakten Überblick über Klassen und deren Abhängigkeiten, skaliert aber nicht gut. Bei der Betrachtung von sehr großen Bundles reicht der Platz auf dem Kreis nicht aus, um alle Knoten ohne Überlappung darzustellen. Eine Möglichkeit wäre es, das Level of Detail anpassbar zu machen, so dass bei Bedarf auf die Granularität von Packages umgeschaltet werden kann.

Anhang

Zur Ausführung der Datenextraktion und -analyse müssen einige Konfigurationen über eine INI-Datei vorgenommen werden (siehe Abschnitt 6.2). Im Folgenden wird beispielhaft eine solche Konfigurationsdatei gezeigt. Diese wurde bei der Evaluierung mit RCE (Abschnitt 6) verwendet:

```
[Project]
name = Remote Component Environment
bundleSourceFolder = src/main/java
bundleRoot = ../rce-trunk/
bundlePaths = de.rcenvironment.*
bundlePathsExclude = \
  de.rcenvironment.*.feature, \
  de.rcenvironment.*.tests, \
  de.rcenvironment.core.gui.workflow, \
  de.rcenvironment.core.component.integration, \
  de.rcenvironment.components.script.execution.jython, \
  de.rcenvironment.components.outputwriter.gui, \
  de.rcenvironment.components.script.common, \
  de.rcenvironment.components.optimizer.gui, \
  de.rcenvironment.components.parametricstudy.gui,

[Storage]
folder = save
filename = rce
extension = model

[VersionControl]
workingCopy = ../rce-trunk/

[PasswordCredential]
user = ;<svn-user>
password = ;<svn-password>

; Alternative: Authentication with SSH-Keyfile
;[SSHCredential]
;user =
;sshKeyFile =
;keyPassphrase =
```

Literaturverzeichnis

[1] A. Abran. *Software Metrics and Software Metrology*. Wiley-IEEE Computer Society Pr, 2010.

[2] O. Alliance. *OSGi Compendium Specification*. OSGi Alliance, 2014.

[3] O. Alliance. *OSGi Core Specification*. OSGi Alliance, 2014.

[4] M. Balzer und O. Deussen. „Level-of-detail visualization of clustered graph layouts“. In: *Visualization, 2007. APVIS'07. 2007 6th International Asia-Pacific Symposium on*. IEEE. 2007, S. 133–140.

[5] P. Baumann. *Software-Bewertung: Ein semantischer Ansatz für Infomationsmaße*. Bd. 299. Informatik-Fachberichte. Springer, 1992.

[6] F. Beck und S. Diehl. „On the Congruence of Modularity and Code Coupling“. In: *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*. ESEC/FSE '11. ACM, 2011, S. 354–364.

[7] J. Bertin. *Graphische Semiologie: Diagramme, Netze, Karten*. Berlin [etc.]: De Gruyter, 1974.

[8] M. Blech u. a. „Keeping Design Documentation Updated through Synchronization of Use-Case-Maps with Implementation“. In: *Argentine Symposium on Software Engineering*. 2006.

[9] E. Bruneton, R. Lenglet und T. Coupaye. „ASM: A code manipulation tool to implement adaptable systems“. In: *In Adaptable and extensible component systems*. 2002.

[10] H. Byelas und A. Telea. „Visualizing metrics on areas of interest in software architecture diagrams“. In: *2009 IEEE Pacific Visualization Symposium*. 2009, S. 33–40.

[11] P. Caserta und O. Zendra. „Visualization of the Static Aspects of Software: A Survey“. In: *Visualization and Computer Graphics, IEEE Transactions on* 17.7 (Juli 2011), S. 913–933.

[12] A. H. Caudwell. „Gource: Visualizing Software Version Control History“. In: *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*. OOPSLA '10. ACM, 2010, S. 73–74.

[13] G. Charters u. a. *Best practices for developing and working with OSGi applications*. Juli 2010.

[14] S. Chidamber und C. Kemerer. „A metrics suite for object oriented design“. In: *Software Engineering, IEEE Transactions on* 20.6 (Juni 1994), S. 476–493.

[15] M. Dahm. „Byte Code Engineering“. In: *Java-Informations-Tage*. 1999, S. 267–277.

[16] S. Diehl. *Software visualization: visualizing the structure, behaviour, and evolution of software*. Springer Science & Business Media, 2007.

[17] A. Elhag und R. Mohamad. „Metrics for evaluating the quality of service-oriented design“. In: *Software Engineering Conference (MySEC), 2014 8th Malaysian*. Sep. 2014, S. 154–159.

[18] N. E. Fenton. *Software Metrics: A Rigorous Approach*. London, UK, UK: Chapman & Hall, Ltd., 1991.

[19] B. Fry. *Visualizing data: Exploring and explaining data with the processing environment*. O'Reilly Media, Inc., 2007.

[20] G. W. Furnas. „Generalized Fisheye Views“. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '86. Boston, Massachusetts, USA: ACM, 1986, S. 16–23.

[21] Y. Ghanam und S. Carpendale. „A Survey Paper on Software Architecture Visualization“. In: (2015).

[22] J. Gosling u. a. *The Java Language Specification, Java SE 7 Edition*. 1st. Addison-Wesley Professional, 2013.

[23] D. Gračanin, K. Matković und M. Eltoweissy. „Software visualization“. In: *Innovations in Systems and Software Engineering* 1.2 (2005), S. 221–230.

[24] H. Graham, H. Y. Yang und R. Berrigan. „A Solar System Metaphor for 3D Visualisation of Object Oriented Software Metrics“. In: *Proceedings of the 2004 Australasian Symposium on Information Visualisation - Volume 35*. APVis '04. Australian Computer Society, Inc., 2004, S. 53–59.

[25] S. Hamza, S. Sadou und R. Fleurquin. „Measuring Qualities for OSGi Component-Based Applications“. In: *Quality Software (QSIC), 2013 13th International Conference on*. Juli 2013, S. 25–34.

[26] N. Hawes, S. Marshall und C. Anslow. „CodeSurveyor: Mapping Large-Scale Software to Aid in Code Comprehension“. In: *Proc. VISSOFT*. IEEE, 2015, S. 96–105.

[27] D. Holten, R. Vliegen und J. J. van Wijk. „Visual Realism for the Visualization of Software Metrics“. In: *3rd IEEE International Workshop on Visualizing Software for Understanding and Analysis*. 2005, S. 1–6.

[28] D. Holten. „Hierarchical Edge Bundles: Visualization of Adjacency Relations in Hierarchical Data“. In: *IEEE Transactions on Visualization and Computer Graphics* 12.5 (Sep. 2006), S. 741–748.

[29] D. Holten und J. J. van Wijk. „Visual Comparison of Hierarchically Organized Data“. In: *Proceedings of the 10th Joint Eurographics / IEEE - VGTC Conference on Visualization*. EuroVis'08. The Eurographs Association John Wiley Sons, Ltd., 2008, S. 759–766.

[30] K. M. S. J. G. Williams und E. Morse. „Visualization“. In: *Annual Review of Information Science and Technology (ARIST)* 30 (1995), 161 207.

[31] B. Johnson und B. Shneiderman. „Tree-Maps: A Space-filling Approach to the Visualization of Hierarchical Information Structures“. In: *Proceedings of the 2Nd Conference on Visualization '91*. VIS '91. IEEE Computer Society Press, 1991, S. 284–291.

[32] *VISSOFT*. Bremen, Germany. 2015.

[33] C. Knight und M. Munro. „Comprehension with[in] virtual environment visualisations“. In: *Program Comprehension, 1999. Proceedings. Seventh International Workshop on*. 1999, S. 4–11.

[34] J. Lakos. *Large-scale C++ Software Design*. Addison Wesley Longman Publishing Co., Inc., 1996.

[35] P. Lam, E. Bodden und L. Hendren. *The Soot framework for Java program analysis: a retrospective*. 2009.

[36] M. Lanza u. a. „CodeCrawler - an information visualization tool for program comprehension“. In: *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.* Mai 2005, S. 672–673.

[37] C. Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process*. 2nd. Prentice Hall PTR, 2001.

[38] S. Liang und G. Bracha. „Dynamic Class Loading in the Java Virtual Machine“. In: *Proceedings of the 13th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*. OOPSLA '98. ACM, 1998, S. 36–44.

[39] T. Lindholm u. a. *The Java Virtual Machine Specification, Java SE 7 Edition*. 1st. Addison-Wesley Professional, 2013.

[40] J. I. Maletic, A. Marcus und M. L. Collard. „A Task Oriented View of Software Visualization“. In: *Proceedings of the 1st International Workshop on Visualizing Software for Understanding and Analysis*. VISSOFT '02. IEEE Computer Society, 2002.

[41] A. Marcus, L. Feng und J. I. Maletic. „3D Representations for Software Visualization“. In: *Proceedings of the 2003 ACM Symposium on Software Visualization*. SoftVis '03. San Diego, California: ACM, 2003.

[42] R. Marinescu. „Detection strategies: metrics-based rules for detecting design flaws“. In: *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on*. Sep. 2004, S. 350–359.

[43] R. C. Martin. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall, 2003.

[44] A. von Mayrhauser und A. M. Vans. „Comprehension Processes During Large Scale Maintenance“. In: *Proceedings of the 16th International Conference on Software Engineering*. ICSE '94. IEEE Computer Society Press, 1994, S. 39–48.

[45] A. von Mayrhauser, A. M. Vans und A. E. Howe. „Program understanding behaviour during enhancement of large-scale software“. In: *Journal of Software Maintenance: Research and Practice* 9.5 (1997), S. 299–327.

[46] J. McAffer, J.-M. Lemieux und C. Aniszczyk. *Eclipse Rich Client Platform*. 2nd. Addison-Wesley Professional, 2010.

[47] J. McAffer, P. VanderLei und S. Archer. *OSGi and Equinox: Creating Highly Modular Java Systems*. 1st. Addison-Wesley Professional, 2010.

[48] M. Mikowski und J. Powell. *Single Page Web Applications: JavaScript End-to-end*. 1st. Manning Publications Co., 2013.

[49] S. Murray. *Interactive Data Visualization for the Web*. O'Reilly Media, Inc., 2013.

[50] G. J. Myatt und W. P. Johnson. *Making Sense of Data III: A Practical Guide to Designing Interactive Data Visualizations*. Wiley Publishing, 2011.

[51] B. A. Myers. „Taxonomies of visual programming and program visualization“. In: *Journal of Visual Languages & Computing* 1.1 (1990), S. 97–123.

[52] V. Nguyen u. a. „A SLOC Counting Standard“. In: *COCOMO II Forum 2007*. 2007.

[53] S. Northover und M. Wilson. *Swt: The Standard Widget Toolkit, Volume 1*. First. Addison-Wesley Professional, 2004.

[54] C. O'Reilly, D. Bustard und P. Morrow. „The War Room Command Console: Shared Visualizations for Inclusive Team Coordination“. In: *Proceedings of the 2005 ACM Symposium on Software Visualization*. SoftVis '05. ACM, 2005, S. 57–65.

[55] T. Panas, R. Berrigan und J. Grundy. „A 3D Metaphor for Software Production Visualization“. In: *Proceedings of the Seventh International Conference on Information Visualization*. IV '03. IEEE Computer Society, 2003.

[56] T. Panas u. a. „Communicating Software Architecture using a Unified Single-View Visualization“. In: *Engineering Complex Computer Systems, 2007. 12th IEEE International Conference on*. Juli 2007, S. 217–228.

[57] G. Parker, G. Franck und C. Ware. „Visualization of large nested graphs in 3D: Navigation and interaction“. In: *Journal of Visual Languages & Computing* 9.3 (1998), S. 299–317.

[58] R. Pawlak u. a. „Spoon: A Library for Implementing Analyses and Transformations of Java Source Code“. In: *Software: Practice and Experience* (2015).

[59] M. Perepletchikov u. a. „Coupling Metrics for Predicting Maintainability in Service-Oriented Designs“. In: *Software Engineering Conference, 2007. ASWEC 2007. 18th Australian*. Apr. 2007, S. 329–340.

[60] M. Petre. „Mental Imagery, Visualisation Tools and Team Work“. In: *Second Program Visualization Workshop*. 2002.

[61] M. Petre, A. F. Blackwell und T. R. G. Green. „Software Visualization“. In: Hrsg. von J. T. Stasko, M. H. Brown und B. A. Price. MIT Press, 1997. Kap. Cognitive Questions in Software Visualisation, S. 453–480.

[62] M. Petre und E. de Quincey. „A gentle overview of software visualisation“. In: *Psychology of Programming Interest Group (PPIG)* (Sep. 2006).

[63] R. Pressman. *Software Engineering: A Practitioner's Approach*. 7. Aufl. McGraw-Hill, Inc., 2010.

[64] A. Rajaraman und J. D. Ullman. *Mining of Massive Datasets*. Cambridge University Press, 2011.

[65] *RCE User Guide*. 5.2.1. Deutsches Zentrum für Luft- und Raumfahrt. 2015.

[66] R. Reussner. *Handbuch der Software-Architektur*. Hrsg. von W. Hasselbring. 2. Aufl. dpunkt Verlag, 2009.

[67] J. Salasin und A. M. Madni. „Metrics For Service-Oriented Architecture (SOA) Systems: What Developers Should Know“. In: *J. Integr. Des. Process Sci.* 11.2 (Apr. 2007), S. 55–71.

[68] C. R. d. Santos u. a. „Metaphor-Aware 3D Navigation“. In: *Proceedings of the IEEE Symposium on Information Vizualization 2000*. INFOVIS '00. IEEE Computer Society, 2000.

[69] D. C. Schmidt. „Model-driven engineering“. In: *COMPUTER-IEEE COMPUTER SOCIETY-* 39.2 (2006), S. 25.

[70] I. Sommerville. *Software Engineering (7th Edition)*. Pearson Addison Wesley, 2004.

[71] D. Steinberg u. a. *EMF: Eclipse Modeling Framework 2.0*. 2nd. Addison-Wesley Professional, 2009.

[72] M.-A. D. Storey, K. Wong und H. A. Muller. „How Do Program Understanding Tools Affect How Programmers Understand Programs“. In: *Proceedings of the Fourth Working Conference on Reverse Engineering (WCRE '97)*. WCRE '97. IEEE Computer Society, 1997.

[73] A. L. Tavares und M. T. Valente. „A Gentle Introduction to OSGi“. In: *SIGSOFT Softw. Eng. Notes* 33.5 (Aug. 2008).

[74] A. C. Telea. *Data Visualization: Principles and Practice, Second Edition*. 2nd. A. K. Peters, Ltd., 2014.

[75] M. Termeer u. a. „Visual Exploration of Combined Architectural and Metric Information“. In: *3rd IEEE International Workshop on Visualizing Software for Understanding and Analysis*. 2005, S. 1–6.

[76] H. Trauboth. *Software-Qualitätssicherung. Konstruktive und analytische Maßnahmen*. 7. Aufl. Oldenbourg, 1996.

[77] C. Ullenboom. *Java ist auch eine Insel: Das umfassende Handbuch (Galileo Computing)*. 10. Aufl. 2010.

[78] C. Walls. *Modular Java: Creating Flexible Applications with Osgi and Spring*. 1st. Pragmatic Bookshelf, 2009.

[79] M. Ward, G. Grinstein und D. Keim. *Interactive Data Visualization: Foundations, Techniques, and Applications*. 2. Aufl. A. K. Peters, Ltd., 2010.

[80] A. I. Wasserman und S. Gutz. „The Future of Programming“. In: *Commun. ACM* 25.3 (März 1982), S. 196–206.

[81] R. Wettel und M. Lanza. „Visual Exploration of Large-Scale System Evolution“. In: *2008 15th Working Conference on Reverse Engineering*. Okt. 2008, S. 219–228.

[82] G. Wütherich u. a. *Die OSGi Service Platform: Eine Einführung mit Eclipse Equinox*. Hrsg. von dpunkt.Verlag. 1. Aufl. dpunkt, 2008.

[83] J. S. Yi u. a. „Toward a Deeper Understanding of the Role of Interaction in Information Visualization“. In: *IEEE Transactions on Visualization and Computer Graphics* 13.6 (Nov. 2007), S. 1224–1231.

[84] B. Zhang. „Computer vision vs. human vision“. In: *Cognitive Informatics (ICCI), 2010 9th IEEE International Conference on*. Juli 2010, S. 3–3.

Website-Verzeichnis

[85] *Asynchronous Module Definition (AMD) Specification*. URL: https://github.com/amdjs/amdjs-api/wiki/AMD (besucht am 07. 04. 2016).

[86] *D3.js - Data-Driven Documents*. URL: https://d3js.org/ (besucht am 13. 04. 2016).

[87] *DLR Simulations- und Softwaretechnik – RCE*. URL: http://www.dlr.de/sc/en/desktopdefault.aspx/tabid-5625/9170_read-17513/ (besucht am 29. 02. 2016).

[88] *Eclipse Java Development Tools (JDT)*. URL: http://www.eclipse.org/jdt/ (besucht am 26. 03. 2016).

[89] *EMF Sever and Storage*. URL: https://eclipse.org/modeling/server.php (besucht am 26. 03. 2016).

[90] *emfjson*. URL: http://emfjson.org/ (besucht am 26. 03. 2016).

[91] *Equinox*. URL: http://www.eclipse.org/equinox/ (besucht am 18. 04. 2016).

[92] *FreePic*. URL: http://www.freepik.com/ (besucht am 22. 04. 2016).

[93] *Gource - a software version control visualization tool*. URL: http://gource.io/ (besucht am 15. 04. 2016).

[94] *JavaHL*. URL: http://subclipse.tigris.org/wiki/JavaHL (besucht am 25. 03. 2016).

[95] *JDT Core Component – Eclipse*. URL: https://eclipse.org/jdt/core/ (besucht am 26. 03. 2016).

[96] *Manifest (Java Platform SE 7)*. URL: https://docs.oracle.com/javase/7/docs/api/java/util/jar/Manifest.html (besucht am 27. 03. 2016).

[97] *ManifestElement (Eclipse Platform API Specification)*. URL: http://help.eclipse.org/mars/index.jsp?topic=/org.eclipse.platform.doc.isv/reference/api/org/eclipse/osgi/util/package-summary.html (besucht am 27. 03. 2016).

[98] *PDE Incubator Dependency Visualization (Eclipse Plugin)*. URL: https://marketplace.eclipse.org/content/pde-incubator-dependency-visualization (besucht am 15. 04. 2016).

[99] *Remote Access API – Jenkins*. URL: https://wiki.jenkins-ci.org/display/JENKINS/Remote+access+API (besucht am 17. 04. 2016).

[100] *RequireJS*. URL: http://requirejs.org/ (besucht am 07. 04. 2016).

[101] *SOAP API – Mantis Bugtracker*. URL: `https://www.mantisbt.org/manual/admin.config.soap.html` (besucht am 17. 04. 2016).

[102] *Spoon - Source Code Analysis and Transformation for Java*. URL: `http://spoon.gforge.inria.fr/` (besucht am 26. 03. 2016).

[103] *SVN Client Adapter – Subclipse*. URL: `http://subclipse.tigris.org/svnClientAdapter.html` (besucht am 25. 03. 2016).

[104] *SVNKit – Subversion for Java*. URL: `http://svnkit.com/` (besucht am 25. 03. 2016).

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

Abkürzung	Bedeutung
AMD	Asynchronous Module Definition
API	Application Programming Interface
CSS	Cascading Style Sheets
DLR	Deutsches Zentrum für Luft- und Raumfahrt
DOM	Document Object Model
DS	Declarative Service
EMF	Eclipse Modeling Framework
FPA	Function Point Analysis
GUI	Graphical User Interface
HTML	Hypertext Markup Language
IDE	Integrated Developlment Environment
JDT	Java Development Tools
JNI	Java Native Interface
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
LOC	Lines of Code
LOCM	Lack of Cohesion in Methods
POJO	Plain Old Java Object
RCE	Remote Component Environment
RCP	Rich Client Platform
SPA	Single Page Application
SSH	Secure Shell
SVG	Scalable Vector Graphics
SWT	Standard Widget Toolkig
Tf-idf	Term Frequency - Inverse Document Freqency
UML	Unified Modeling Language
WMC	Weighted Methods per Class
XMI	XML Metadate Interchagen
XML	Extensible Markup Language

Eidesstattliche Versicherung

_____ _____

Name, Vorname Matr.-Nr.

Ich versichere hiermit an Eides statt, dass ich die vorliegende Bachelorarbeit/Masterarbeit* mit dem Titel

_____

_____

_____

selbstständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate kenntlich gemacht. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

_____ _____

Ort, Datum Unterschrift

*Nichtzutreffendes bitte streichen

Belehrung:

Wer vorsätzlich gegen eine die Täuschung über Prüfungsleistungen betreffende Regelung einer Hochschulprüfungsordnung verstößt, handelt ordnungswidrig. Die Ordnungswidrigkeit kann mit einer Geldbuße von bis zu 50.000,00 € geahndet werden. Zuständige Verwaltungsbehörde für die Verfolgung und Ahndung von Ordnungswidrigkeiten ist der Kanzler/die Kanzlerin der Technischen Universität Dortmund. Im Falle eines mehrfachen oder sonstigen schwerwiegenden Täuschungsversuches kann der Prüfling zudem exmatrikuliert werden. (§ 63 Abs. 5 Hochschulgesetz - HG -)

Die Abgabe einer falschen Versicherung an Eides statt wird mit Freiheitsstrafe bis zu 3 Jahren oder mit Geldstrafe bestraft.

Die Technische Universität Dortmund wird gfls. elektronische Vergleichswerkzeuge (wie z.B. die Software „turnitin“) zur Überprüfung von Ordnungswidrigkeiten in Prüfungsverfahren nutzen.

Die oben stehende Belehrung habe ich zur Kenntnis genommen:

_____ _____

Ort, Datum Unterschrift