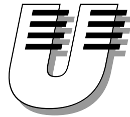

Fachbereich 4: Informatik

Empirische Untersuchung der Frameworkiness von Open–Source–Projekten

Bachelorarbeit

zur Erlangung des Grades eines Bachelor of Science
im Studiengang Informatik

vorgelegt von

Jan Baltzer

Erstgutachter: Prof. Dr. R. Lämmel
Institut für Softwaretechnik

Zweitgutachter: Ekaterina Pek
Institut für Informatik

Koblenz, im Juni 2011

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

	Ja	Nein
Mit der Einstellung der Arbeit in die Bibliothek bin ich einverstanden.	☐	☐
Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.	☐	☐

. . . .

(Ort, Datum) (Unterschrift)

Zusammenfassung

Diese Ausarbeitung implementiert eine Untersuchung von APIs und knüpft dabei an bestehende Analysen wie [**?**] an. Sie grenzt sich von diesen dadurch ab, dass sie mit der Fokussierung auf Frameworks eine neue Betrachtungsweise entwickelt und neben einer reinen *usage analysis* auch den Aufbau von Programmierschnittstellen in die Betrachtung mit einbezieht. Zielsetzung ist es dabei, zunächst alle Aspekte, welche im Zusammenhang mit Frameworks stehen, sog. *Frameworkiness*, zu definieren, diese umzusetzen und die Resultate auszuwerten.

Die Implementation ist dabei in das bestehende Analyse–System des Software Language Teams der Universität Koblenz–Landau eingebettet und konzentriert sich auf Sourcecode–basierte Erweiterungen und Implementierungen. Andere Ansätze, wie die Nutzung von Annotations oder XML–Konfigurationsdateien, werden nicht gezielt weiterverfolgt. Die genannte Umgebung liefert Informationen über Deklarationen und Aufrufe verschiedenster Art, codiert als Prolog–Fakten. Die Umsetzung der Analyse umfasst somit die Transformation dieser Menge unter den Gesichtspunkten der *Frameworkiness*.

Die Untersuchungen weisen im analysierten Software–Corpus die Verwendung von APIs als Frameworks eindeutig nach, wobei insbesondere die Domänen GUI und XML in dieser Hinsicht vermehrt auftreten. Dabei ist zu beobachten, dass sich diese Nutzung auf einzelne Features und Bereiche konzentriert, wie die Ereignisbehandlung in der API `AWT`. Obwohl sich Gebrauch und Konzeption einer API nicht decken müssen, liefert der Aufbau von Programmierschnittstellen interessante Erkenntnisse, da diese zu einem gewissen Teil aus Typen bestehen, die innerhalb der API keinen konkreten Subtyp besitzen. Diese API–Bestandteile können als Indikatoren für einen Entwurf als Framework aufgefasst werden, was jedoch nicht zur Folge hat, dass diese im Anwendungscode erweitert oder implementiert werden. Es wird gezeigt, dass zum einen auch konkrete Klassen als Vorlagen für Framework–Nutzung dienen, zum anderen Komponenten der Java–Laufzeitumgebung und andere APIs als Provider fungieren, die Programmierschnittstellen mit Funktionalität versehen.

APIs finden demnach als Framework Verwendung und lassen dies bis zu einem gewissen Grad auch aus ihrem Aufbau schließen. Offen bleibt, ob der Anwender diese Framework–Fähigkeit ausnutzt, oder aber (auch unbewusst) auf eine Provider–Implementation zurückgreift.

Abstract

This bachelor thesis provides an analysis of APIs and extends existing investigations like [**?**]. It differs from them by the fact that it develops a new perspective by focusing on frameworks. In addition, not only the pure usage is considered, but also the structure of the programming interface. The main objectives are the definitions, realizations and evaluations of all aspects related to frameworks, so–called *frameworkiness*.

The implementation is embedded in the existing system for software analysis of the Software Language Team at the University Koblenz–Landau and focuses on source code based extensions and implementations. Other approaches, such as the use of annotations or XML configuration files, are not investigated specifically. The mentioned environment provides information about declarations and invocations of all kinds and encodes them as Prolog facts. Therefore, the analysis includes the transformation of this particular set with regard to the *frameworkiness*.

The investigations prove the usage of APIs as frameworks, with the domains GUI and XML being most common. It can be observed that this use is concentrated on certain features and areas such as event handling in the `AWT` API. Although usage and conception of an API do not have to coincide, interesting insights can be obtained from the structure of programming interfaces, because they consist to some extent of types, which do not have any concrete subtype within the API. This components can be regarded as indicators of an intention as a framework, but this does not mean that they are extended or implemented in the application's code. It is shown that concrete classes can serve as templates for framework usage likewise and that components of the JAVA Runtime Environment or other programming interfaces act as providers on the other hand, supplying functionality to the APIs.

According to that, APIs are used as frameworks and this fact is shown up to a certain degree from their structure. The question remains, whether the user exploits the framework–ability of a programming interface or falls back (even unknowingly) upon a provider's implementation.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einleitung

1.1 Motivation

Die Wiederverwendung von Software ist ein wesentlicher Leitgedanke in der Programmierung. Frameworks eignen sich dafür, wie Craig Larman in [**?**, S. 539] und Ralph Johnson und Brian Foote in [**?**, o. S.] herausstellen, in hohem Maße. Vor allem in der objektorientierten Programmierung gehört die Verwendung von Frameworks zum Alltag und wird in vielen Softwareprojekten praktiziert. Durch eine stetige Weiterentwicklung und Verbesserung solcher Frameworks bzw. deren Programmierschnittstellen (**API** – application programming interface) ist es jedoch notwendig, Ideen und Verfahren für die Migration auf ein anderes Framework zu entwickeln. Dies kann zum einen bei einem Upgrade auf eine neue Version, zum anderen bei einem Wechsel zwischen zwei unterschiedlichen APIs relevant sein.

In diesem Gebiet der Softwaretechnik, sog. API–Migration, existieren inzwischen eine Vielzahl an Untersuchungen u. a. zu verschiedenen Implementierungstechniken, aber auch zum Verständnis über die Verwendung solcher Programmierschnittstellen in Softwareprojekten. Diese letztere API–Analyse konzentriert sich jedoch bisher im Wesentlichen auf die Library–Elemente von APIs, wie zum Beispiel die Nutzung und die Abdeckung der bereitgestellten Features, wovon aber in dieser Ausarbeitung abgewichen und sich mit dem Gebrauch von Frameworks befasst wird. Da dies ein neuer Aspekt in der Untersuchung von Programmierschnittstellen ist, werden grundlegende Mechanismen und Kriterien eigenständig erarbeitet.

Für diese Art der Analyse existieren zwei motivierende Ansätze. Neben der bereits aufgeführten allgemeinen Relevanz von Frameworks weisen aktuelle Umsetzungen von API–Migrationen Probleme auf, die direkt mit Eigenschaften dieser spezi-

ellen Programmierschnittstellen zusammenhängen. Nach Johnson und Foote ist die *Inversion of Control* eine zentrales Charakteristikum und bezeichnet den Wechsel der Programmsteuerung zwischen Anwendung und Framework. Jede Implementierung einer API–Migration muss diese Veränderung im Kontrollfluss ermöglichen, was jedoch in der Praxis keine einfache Aufgabe darstellt.[1] Solche Schwierigkeiten zeigen auf, dass ein besseres Verständnis im Umgang mit APIs, speziell mit Frameworks, erforderlich ist. Diese Arbeit knüpft an die beschriebene Problematik an und erweitert bestehende Untersuchungen zur Verwendung von APIs wie [**?**]. Die Zielsetzung ist dabei, alle Aspekte eines „frameworkhaften" Gebrauchs (im Folgenden als *Frameworkiness* bezeichnet) zu erkennen, analysieren und darzustellen, also beispielsweise die tatsächliche Verwendung in Softwareprojekten zu messen, aber auch zu prüfen, welche strukturelle Möglichkeiten sich durch den Aufbau von Programmierschnittstellen für einen derartigen Gebrauch anbieten.

1.2 Zentrale Fragestellungen

Diese Thesis präsentiert eine Untersuchung dieser *Frameworkiness* für APIs und Softwareprojekte in Java. Zur Messung und Feststellung realer Fakten ist es notwendig, die beiden beschriebenen Aspekte zu charakterisieren. Demnach sind in einem ersten Schritt folgende Leitfragen zu klären:

- Was ist maßgeblich für den Gebrauch einer API als Framework?
- Woran ist zu erkennen, ob eine API als Framework genutzt werden kann bzw. konzipiert ist?

Basierend auf diesen grundsätzlichen Sachverhalten ist es möglich, derartige Abhängigkeiten zwischen Projekten und APIs zu quantifizieren:

- Welche APIs werden als Framework gebraucht?
- Welche APIs sind als Framework gedacht?

Diesen Ergebnissen werden darüber hinaus folgende qualitative Fragestellungen gegenübergestellt:

- Gibt es Faktoren, von denen die Frameworkiness abhängig ist?
- Lassen sich typische Anwendungsbereiche von Frameworks erkennen?

[1] s. [**?**, S. 5]

- Gibt es verschiedene Arten im Gebrauch von Frameworks?
- Unterscheiden sich Verwendung und ursprüngliche Konzeption einer API?

1.3 Beiträge

Diese Ausarbeitung liefert zum einen Beschreibungen, wie die Verwendung von APIs und der Aufbau von Programmierschnittstellen unter Gesichtspunkten der *Frameworkiness* untersucht werden kann. Es werden wesentlichen Metriken vorgestellt und die notwendigen Messgrößen formal und im Sinne einer Umsetzung in Prolog beschrieben. Basierend auf diesen Aspekten werden einige Analysen wie

- der Nachweise von Framework–Verwendung
- der Aufbau einer API
- die Ausnutzung der API–Strukturen

umgesetzt und die gewonnen Ergebnisse und ihre Aussagen diskutiert.

1.4 Aufbau

Ein erster Abschnitt stellt die generelle Methodik vor, indem die Vorgehensweise erläutert wird und die dafür notwendigen Kriterien aufgezeigt werden. Dies beinhaltet zugleich Definitionen und Erläuterungen von Begriffen, aber vor allem verschiedene Ideen zur Entwicklung der *Frameworkiness*. Die Umsetzung dieser allgemeinen Vorgehensweise behandelt ein folgendes Kapitel, welches aufbauend auf der Beschreibung der Umgebung zur API–Analyse die konkreten Messgrößen erklärt. Abgeschlossen wird diese Arbeit mit der Darstellung und Bewertung der Resultate.

Kapitel 2

Begriffe und Definitionen

Die programmatischen Fragestellungen (s. **??**) beschreiben im Wesentlichen bereits die generelle Methodik, welche jedoch für eine Implementierung weiter zu spezifizieren ist. Dazu ist es allerdings notwendig die Bedeutung bestimmter Begriffe zu erläutern, um an Hand dieser wichtige Kriterien zu definieren.

So enthält die Einleitung dieser Arbeit bereits einige Fachwörter, obwohl diese zur einheitlichen und eindeutigen Verwendung zu definieren sind. Beispielsweise werden APIs bereits in Library und Framework unterteilt, ohne zu klären, wie diese Unterscheidung zustande kommt.

2.1 Framework vs. Library

Genau diese Abgrenzung und die Bedeutung von Frameworks ist für diese Ausarbeitung jedoch von enormer Bedeutung. An dieser Stelle muss aber darauf hingewiesen werden, dass der Begriff „Framework“ keineswegs eindeutig benutzt wird. Diese Ausarbeitung orientiert sich dabei an der Betrachtungsweise, wie sie in [**?**, S. 11] definiert und dargestellt wird:

> A software library contains functions or routines that your application can invoke. A framework, on the other hand, provides generic, cooperative components that your application extends to provide a particular set of functions.

Diese Differenzierung wird in der Abbildung **??** zusammenfassend dargestellt: auf der einen Seite die Verwendung eines Frameworks, in dessen Funktionalität sich über

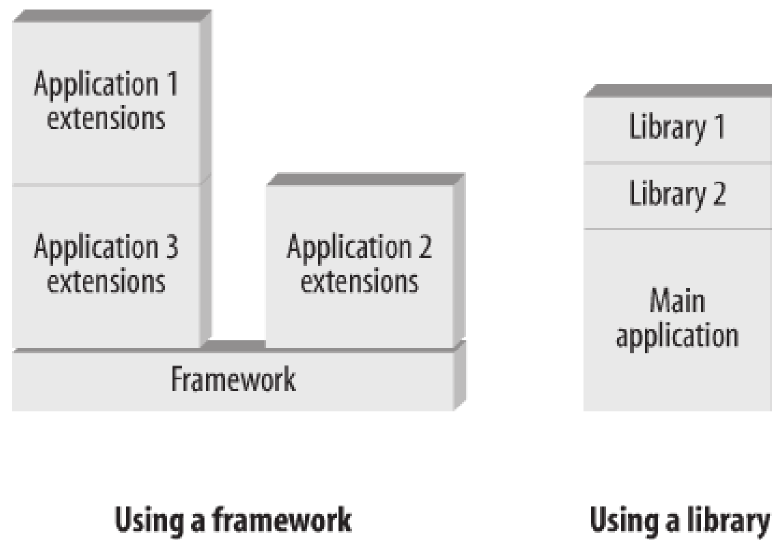

Abbildung 2.1: Abgrenzung zwischen Framework und Library [?, S. 11]

Erweiterungen „eingeklinkt“ wird, auf der anderen Seite der aufrufende Gebrauch einer Library. An dieser Stelle wird neben dem Prinzip der Erweiterung auch die bereits erwähnte *Inversion of Control* (s. **??**) bzw. das *Hollywood Principle* (vgl. [**?**]), welches u. a. in [**?**, S. 539] als populäres Synonym aufgegriffen wird, deutlich: die Basis und somit den Hauptbestandteil bildet das Framework selbst und nicht die eigene Anwendung.

Dieses spezielle Unterscheidungsmerkmal eröffnet den Schwerpunkt dieser Arbeit, welcher auf Frameworks liegt, die über Vererbungen und Schnittstellenimplementierungen im Sourcecode eingebunden werden. Für andere Arten wie die auf Annotations basierten Umsetzungen in den Frameworks `JUnit`, `Hibernate` etc. existieren daher keine speziellen Analyseansätze.

2.2 Relevante Sprachkonstrukte

Da in dieser Arbeit Java–Projekte examiniert werden, stellt sich die Frage, welche Punkte in dieser Programmiersprache als Erweiterungen zu zählen sind. Im Wesentlichen gibt es vier verschiedene Arten:

1. Erweiterungsvererbung
2. Implementierungsvererbung
3. Überschreiben von Methoden

4. Implementieren von Methoden

An Hand von Szenario **??** soll diese Unterscheidung im Folgenden exemplarisch aufgezeigt werden. [1]

```
public interface Swimmable {
  public void swim();
}

public interface Divable extends Swimmable{
  public void dive();
}

public class Watercraft implements Swimmable{
  public void swim() {
    System.out.println("swimming...");
  }
}

public class Ship extends Watercraft {
  public void swim() {
    System.out.println("swimming on water...");
  }
}

public class Submarine implements Divable{
  public void dive() {
    System.out.println("diving...");
  }

  public void swim() {
    System.out.println("swimming...");
  }
}
```

Listing 2.1: Arten von Erweiterungen in Java

2.2.1 Erweiterungsvererbung

Für Hierarchiebeziehungen zwischen Typen existieren zwei Schlüsselworte. Durch `extends` wird eine Erweiterungsvererbung angezeigt, was in zwei Fällen vorkommen kann:

[1] Formale Beschreibungen können [**?**] entnommen werden

- Eine Klasse wird aus einer anderen Klasse abgeleitet.
  Beispiel: **`class`** `Ship` **`extends`** `Watercraft`
  Man verwendet folgende Bezeichnungen:

	Endpunkt 1	Beziehung	Endpunkt 2
	derived class oder *subclass*	*super*	*base class* oder *superclass*
Beispiel	`Ship`	**`class extends`**	`Watercraft`

Tabelle 2.1: Erweiterungsvererbung - Benennung bei Klassen

- Ein Interface wird aus einem anderen Interface abgeleitet.
  Beispiel: **`interface`** `Divable` **`extends`** `Swimmable`
  Nachstehende Tabelle zeigt, wie die Benennung erfolgt:

	Endpunkt 1	Beziehung	Endpunkt 2
	derived interface oder *subinterface*	*extends*	*base interface* oder *superinterface*
Beispiel	`Divable`	**`interface extends`**	`Swimmable`

Tabelle 2.2: Erweiterungsvererbung - Benennung bei Interfaces

Dabei gilt, dass eine Klasse maximal eine andere Klasse erweitern kann, während ein Interface durchaus aus mehreren anderen Interfaces abgeleitet sein kann.

2.2.2 Implementierungsvererbung

Das andere Schlüsselwort (**`implements`**) zeigt Implementierungsvererbungen an. Die typische Erscheinungsform liegt vor, wenn eine Klasse ein Interface implementiert.
Beispiel: **`class`** `Watercraft` **`implements`** `Swimmable`
Die Tabelle **??** zeigt dabei die Begriffe, die für Implementierungsvererbungen im Gebrauch sind. In Java gilt, dass eine Klasse beliebig viele Interfaces implementieren kann.

	Endpunkt 1	Beziehung	Endpunkt 2
	derived class oder *subclass*	*implements*	*base interface* oder *superinterface*
Beispiel	`Watercraft`	**`implements`**	`Swimmable`

Tabelle 2.3: Implementierungsvererbung - Benennung

2.2.3 Überschreiben von Methoden

Je nachdem wie zwei Typen in hierarchischer Beziehung zueinanderstehen, werden fallweise vererbte Methoden, die im Subtyp neu implementiert werden, unterschieden. Falls es sich um eine Erweiterungsvererbung handelt und die Methode des Supertyps nicht abstrakt ist, bezeichnet man dies als Überschreiben. In Falle von Szenario **??** überschreibt beispielsweise die Methode `swim` in `Ship` die Methode `swim` der Klasse `Watercraft`.

Im Allgemeinen werden Überschreibungen nochmals unterschieden, je nachdem ob es sich um eine statische Methode oder um eine Instanz–Methode handelt. Da diese Differenzierung hinsichtlich der Erweiterung an sich keinen Mehrwert liefert, wird dies jedoch nicht weiter verfolgt.

2.2.4 Implementieren von Methoden

Handelt es sich jedoch um eine Implementierungsvererbung oder tritt der bereits angedeutete Fall ein, dass eine abstrakte Methode betroffen ist, liegt eine Implementierung vor. So implementiert im obigen Beispiel (vgl. Codebeispiel **??**) die Methode `swim` in `Watercraft` die entsprechende Methode des Interface `Swimmable`.

2.3 Aspekte der Frameworkiness

Mit Hilfe dieser Erläuterungen lässt sich die *Frameworkiness* gezielter beschreiben und dabei vor allem die charakterisierenden Kriterien herausstellen, die somit notwendigerweise zu extrahieren sind.

Für die folgenden Abschnitte gelten nachstehende Bezeichnungen und Abkürzungen:

- Sei $\mathbb{A}$ die Menge aller APIs: $\mathbb{A} = \{a_1, a_2, ..., a_m\}$
- v_a bezeichnet den entsprechenden Wert v der API a
  Beispiel: $\textit{base type}_a$ bezeichnet die Anzahl an Supertypen in der API a, die von Projekten erweitert oder implementiert werden.

2.3.1 Subtyping coverage

Auf der ein Seite ist zu untersuchen, wie eine API verwendet wird. Dazu sind verschiedene Softwareprojekte zu betrachten und Informationen über den direkten Gebrauch der Programmierschnittstelle zu sammeln. Eine signifikante Größe für eine API stellt dabei der Anteil von erweiterten und implementierten Typen an der Gesamtzahl aller öffentlichen Klassen und Schnittstellen dar. Sie beschreibt, wie die bereitgestellten Typen durch Framework–Verwendung abgedeckt werden.

$$coverage_a = \frac{distinct\ base\ types_a}{public\ types_a}, a \in \mathbb{A}$$

2.3.2 Subtyping requiredness

Diesem bloßen Gebrauch einer API ist deren funktionale Struktur gegenüberzustellen. Dies beinhaltet die Analyse der API hinsichtlich der Fragestellung, inwieweit diese als Framework konzipiert ist. Das wesentliche Merkmal hierfür ist, dass abstrakte Typen in einer API vorhanden sind, die nicht entsprechend erweitert oder implementiert werden, sodass konkrete Subklassen innerhalb der API vorhanden sind. Das Fehlen solcher Unterklassen impliziert die These, dass gerade diese Typen in den Applikationen in eine Vererbungshierarchie einzubinden sind, da von ihnen kein anderweitiger Nutzen ausgeht. Dies führt zu einer im Folgenden wesentlichen Metrik, der sog. *subtyping requiredness*, welche die Notwendigkeit an Subtyping–Beziehungen angibt und somit eine potentielle Eignung als Framework darstellt.
Sei $non\text{–}suptyped_a := in\ api\ unextended\ abstr.\ classes_a + in\ api\ unimplemented\ interfaces_a$.

$$requiredness_a = \frac{non\text{–}suptyped_a}{types_a}, a \in \mathbb{A}$$

2.3.3 Subtyping inapplicability

Ebenso wie dies ein möglicher Hinweise für eine Verwendbarkeit als Framework ist, existieren jedoch auch Indizes für das Gegenteil. In vielen Programmiersprachen kann

explizit verhindert werden, dass eine konkrete Klasse in irgendeiner Form erweitert wird.[2] Solch eine *sealed class* kann demnach nur als Library–Element Verwendung finden, sodass ein hoher Anteil an derartigen Klassen ein Anhaltspunkt gegen eine Anlage als Framework ist. Dies wird durch die nachstehende Größe der *subtyping inapplicability* beschrieben.

$$inapplicability_a = \frac{sealed\text{-}classes_a}{types_a}, a \in \mathbb{A}$$

2.3.4 Requiredness usage

Ein wichtiger Aspekt ist basierend darauf die Fragestellung, inwieweit die Notwendigkeit an Subtyping–Beziehungen tatsächlich genutzt wird. Dazu ist zu untersuchen, ob die *non–suptyped* Typen einer API in den Projekten erweitert oder implementiert werden. Diese *requiredness usage* gibt somit in einem gewissen Rahmen an, ob die Konzeption einer Programmierschnittstelle der realen Verwendung nahe kommt.

$$usage_a = \frac{distinct\ frameworky\ base\ types_a}{distinct\ base\ types_a}, a \in \mathbb{A},$$

wobei $frameworky\ base\ types$ Erweiterungen und Implementierungen von Typen der Art *non-suptyped* bezeichnet.

[2] In Java geschieht dies durch das Schlüsselwort `final` in der Klassendeklaration

Kapitel 3

Allgemeine Methodik

Primäres Ziel dieser Ausarbeitung ist die Umsetzung der beschriebenen Aspekte. Zunächst bedarf es der Klärung, in welcher Form aus den APIs und Projekten Informationen gewonnen werden können.

Für diese Analyse hat das Software Language Team der Universität Koblenz–Landau im Sommersemester 2010 mit dem Projekt „APIAnalysis2.0“ eine Umgebung geschaffen, die aus mehreren Corpora[1] und Tools zur Untersuchung von Projekten und APIs besteht.

3.1 Projektauswahl

Derzeit wird mit ausgewählten Projekten gearbeitet, von denen „guter“ Programmiercode im Sinne der Objektorientierung zu erwarten ist. Der Analyse und somit auch den Ergebnissen, die in dieser Arbeit vorgestellt werden, liegt der Corpus „Java– Shape“ zu Grunde, der auf den in [**?**] genutzten Projekten basiert.[2] Die Tabelle **??** fasst die Größenordnung dieses Corpus zusammen.

	# number
projects	32
classes	33685
interfaces	3420
methods	243355

Tabelle 3.1: Zusammenfassung Corpus „JavaShape“

[1] Corpus bezeichnet eine Menge an bestimmten Software-Projekten

[2] Im Anhang ist eine Übersicht mit allen Projekten zu finden

3.2 API-Auswahl

Die implementierte Analyse geht von einer festen Menge an APIs aus. Diese umfasst die wichtigsten Programmierschnittstellen der Laufzeitumgebung, welche explizit durch Black- bzw. Whitelisting bestimmter Packages und Typen zusammengefasst werden. Vorlage für diese Einteilung bilden die Spezifikationen der einzelnen APIs in der Java–Dokumentation[3]. Zusätzlich werden jedoch auch diejenigen APIs gezielt analysiert, welche häufig als JAR–Archive in den Projekten auftauchen.[4] Es ist darauf hinzuweisen, dass diese quantitative Auswertung keine qualitative Aussage über die tatsächliche Verwendung einer Programmierschnittstelle ermöglicht. Es ergibt sich zusammengefasst ein API-Pool, wie er in der Tabelle **??** dargestellt ist.

	# number
APIs	75
classes	22456
interfaces	3392
methods	212884

Tabelle 3.2: Zusammenfassung API-Pool

3.3 Tools

Im Rahmen des bereits erwähnten Projekts „APIAnalysis2.0“ wurden auch mehrere Werkzeuge zur Analyse von APIs und Projekten entwickelt. Grundlage bildet wie auch in [**?**, **?**] die Analyse des abstrakten Syntaxbaums (**AST** – abstract syntax tree) mit Hilfe eines Compiler–Plug–ins, welches aufgrund einiger Schwierigkeiten durch das Design einzelner Projekte zur Zeit jedoch keine Verwendung findet. Daher wurden erfolgreich zwei weitere Tools mit dem Ziel der Verfahrensoptimierung entwickelt:

- AST–Analyse mit Hilfe des Frameworks `Recoder`[5]:
  Mit Hilfe dieses Werkzeugs kann der Sourcecode der Projekte untersucht werden.

- Bytecode–Analyse mit Hilfe des Frameworks `ASM`[6]:
  Dieses Tool extrahiert Informationen aus den Programmierschnittstellen. Teil-

[3] vgl. http://download.oracle.com/javase/6/docs/
[4] Eine vollständige Liste der enthaltenen APIs ist im Anhang zu finden
[5] vgl. http://sourceforge.net/projects/recoder/
[6] vgl. http://asm.ow2.org/

weise wird dieses Tool auch auf Projekte angewendet, falls diese nur in binärer Form vorliegen.

Ein wichtiges Grundwerkzeug ist dabei die Analyse des Build–Vorgangs der Projekte, um auf diese Weise die tatsächlichen Quelldateien zu bestimmen, sich somit auf die Kernbereiche zu beschränken und beispielsweise experimentelle und überholte Komponenten auszuschließen.

3.4 Faktenbasis

Alle drei Werkzeuge generieren dieselbe Informationsdarstellung, wobei der Gehalt je nach Anwendungsfall variieren kann. Prinzipiell können

- Klassen–/Interface–Deklaration mit
  vollständigem Packagenamen, Typnamen, vollständigem Namen aller Superklassen und -interfaces, sowie einer Liste aller Modifier
- Methoden–/Konstruktor–Deklaration mit
  vollständigem Typnamen, Methodennamen[7], einer Liste der vollständigen Typnamen aller Attribute, vollständigem Name des Rückgabewerts[8] und einer Liste aller Modifier
- Felddeklaration mit
  vollständigem Namen der enthaltenden Klasse, vollständigem Typnamen, Feldnamen und einer Liste aller Modifier
- Deklaration einer lokalen Variablen mit
  vollständigem Namen der enthaltenden Methode, vollständigem Typnamen, Variablennamen und einer Liste aller Modifier
- Methoden–/Konstruktor–Aufruf mit:
  vollständigem Methodennamen der aufrufenden Methode, vollständigem Methodenname der aufgerufenen Methode, sowie einer Liste der vollständigen Typnamen aller Attribute der aufgerufenen Methode

analysiert und in Prolog–Fakten (siehe Code–Ausschnitt **??**) transformiert werden. Zusätzlich beinhalten alle extrahierten Informationen einen Identifikator zur eindeutigen Zuordnung zu einem Projekt oder einer API.

[7] bei Konstruktoren ist Name `<init>`

[8] bei Konstruktoren ist der Rückgabewert leer

```
1 class(ID,QPackage,Name,QSuper,QImplements,Mods).
2 interface(ID,QPackage,Name,QExtends,Mods).
3 method(ID,QType,Name,QAttrs,QReturn,Mods).
4 field(ID,QContainerClass,QType,Name,Mods).
5 localVariable(ID,QContainerMethod,QType,Name,Mods).
6 methodCall(ID,QCaller,QCallee,QAttrs).
```

Listing 3.1: Ausgangsprädikate

3.5 Konzeptioneller Weg

Die Konzeption der Untersuchung hinsichtlich Framework–Verhalten von APIs stellt die Grafik **??** dar. Basierend auf einer bestimmen Faktenbasis (vgl. (1)) erfolgt die Iteration über die Projekte, d.h. ein einzelner Analyseschritt verarbeitet die Fakten eines Projektes und einer bestimmten Teilmenge der Programmierschnittstellen. Letztere werden jedoch zunächst separat durch einen eigenen *analyzer* untersucht (vgl. (2)). Dort findet die Extraktion von Aspekten statt, die unabhängig von der Verwendung in Projekten ist, wie beispielsweise die Frage nach der Konzeption einer API. Der zweite *analyzer* (vgl. (3)) beinhaltet die Analyse der Projekte. Allgemein werden mittels einer Filterbibliothek gezielte Anfragen an die Faktenbasis gestellt (vgl. (2a) bzw. (3a)) und Resultate ggf. in Prädikate der Ergebnismenge transformiert (vgl. (2b) bzw. (3b)). Die so entstehende Ausgabemenge wird dann unterschiedlich verarbeitet und grafisch aufbereitet (vgl. (4)).

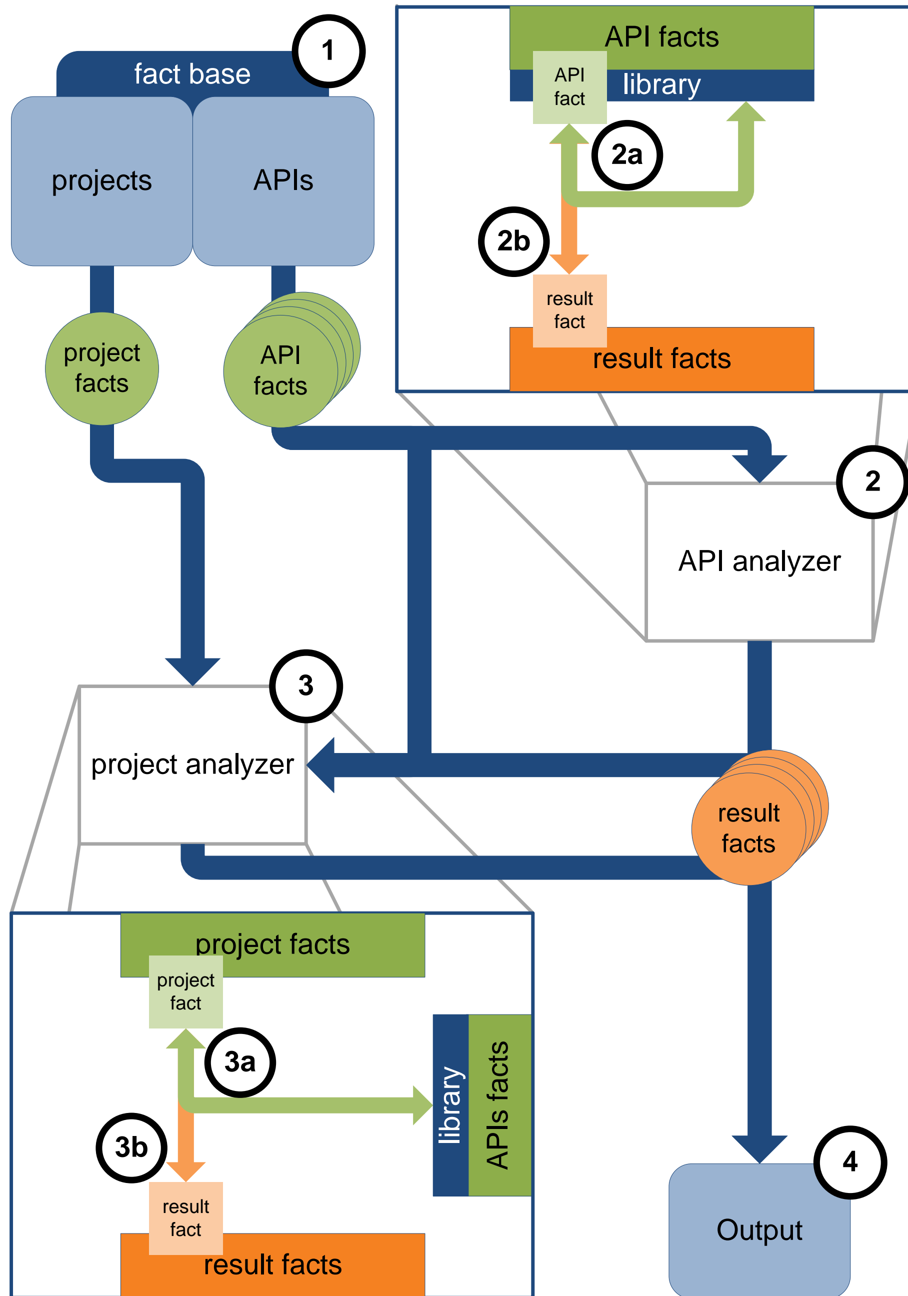

Abbildung 3.1: Konzeptioneller Weg

Kapitel 4

Messgrößen

Auf Grund dieser Umgebung sind viele Implementierungsdetails bereits vorgegeben. So sind die Programmiersprache mit Prolog und der Informationsgehalt der Eingabemenge durch die existierende Faktenbasis definiert. Die Umsetzung umfasst demnach vor allem die Entwicklung einer Ergebnismenge in Form von Prolog–Prädikaten und eines Mechanismus zur Transformation dieser Ausgangsinformationen hin zur Ausgabedarstellung. Im Folgenden werden zur Darstellung dieses Vorgangs die diversen Messgrößen formal durch Prolog–Code beschrieben.

4.1 Beziehungen zwischen Projekten und APIs

4.1.1 Vererbungen

Eine Vererbung heißt genau dann standardmäßig, wenn der betroffene API–Typ innerhalb der Programmierschnittstelle über eine konkrete Implementation verfügt. Eine API–Klasse, welche durch eine Klasse eines Projekts erweitert wird, wird dabei durch das Prädikat `standard_base_class` vollständig beschrieben. Ebenso bezieht sich jedes Prädikat `standard_base_interface` auf ein API–Interface, welches entweder in einem Projekt durch eine Klasse implementiert oder durch ein Interface erweitert wird. Analog werden die abgeleiteten Typen dargestellt, welche die entsprechenden Projekt–Elemente codieren.

Im Gegensatz dazu sind „framework–hafte“ Vererbungen dadurch gekennzeichnet, dass es sich bei der Klasse oder dem Interface der API um einen abstrakten Typ handelt, der innerhalb dieser Programmierschnittstelle nicht konkret implementiert ist. Die formalen Definitionen unterscheiden sich demzufolge nur in zusätzlichen Ei-

genschaften für die betroffenen API–Typen.

Standard base class

```
% standard_base_class(Pid,Aid,Package,Name,Supertypes,Modifiers)
% class extends class
standard_base_class(Pid,Aid,P,N,Ss,Ms) :-
    class(Aid,P,N,S,Is,Ms),
    Ss = [S|Is],
    full_name(P, N, Full),
    class(Pid,_,_,Full,_,_).
```

Standard base interface

```
% standard_base_interface(Pid,Aid,Package,Name,Supertypes,Modifiers)
% class implements interface
standard_base_interface(Pid,Aid,P,N,Ss,Ms) :-
    interface(Aid,P,N,Ss,Ms),
    full_name(P, N, Full),
    class(Pid,_,_,_,Is,_),
    member(Full ,Is).
% interface extends interface
standard_base_interface(Pid,Aid,P,N,Ss,Ms) :-
    interface(Aid,P,N,Ss,Ms),
    full_name(P,N,Full),
    interface(Pid,_,_,Es,_),
    member(Full,Es).
```

Standard derived class

```
% standard_drvd_class(Pid,Aid,Package,Name,Supertypes,Modifiers)
% class extends class
standard_drvd_class(Pid,Aid,P,N,Ss,Ms) :-
    class(Pid,P,N,S,Is,Ms),
    Ss = [S|Is],
    class(Aid,P_a,N_a,_,_,_),
    full_name(P_a, N_a, S).
% class implements interface
standard_drvd_class(Pid,Aid,P,N,Ss,Ms) :-
    class(Pid,P,N,S,Is,Ms),
    Ss = [S|Is],
    interface(Aid,P_a,N_a,_,_),
```

```
    full_name(P_a, N_a, Full),
    member(Full, Is).
```

Standard derived interface

```
% standard_drvd_interface(Pid,Aid,Package,Name,Supertypes,Modifiers)
% interface extends interface
standard_drvd_interface(Pid,Aid,P,N,Ss,Ms) :-
    interface(Pid,P,N,Ss,Ms),
    interface(Aid,P_a,N_a,_,_),
    full_name(P_a, N_a, Full),
    member(Full, Ss).
```

Frameworky base/derived types

```
    ...
    % for classes
    full_name(P,N,Full),
    (   % no subtype
        \+ class(Aid,P_a,N_a,Full,_,M_a)
    ;   % abstract subtype
        class(Aid,P_a,N_a,Full,_,M_a),
        member('abstract', M_a)
    ),
    ...
```

bzw.

```
    ...
    % for interface
    full_name(P,N,Full),
    (   % no subtype
        \+ (class(Aid,P_a,N_a,_,Is_a,M_a), member(Full, Is_a))
    ;   % abstract subtype
        class(Aid,P_a,N_a,_,Is_a,M_2),
        member(Full, Is_a),
        member('abstract', M_a)
    ),
    ...
```

4.1.2 Erweiterungen von Methoden

Die Unterscheidung zwischen standardmäßigen und „framework–haften“ Erweiterungen wird für diesen Teilbereich der Beziehungen nicht vorgenommen. Alle über-

schriebenen und -implementierten Methoden werden als solche ohne weitere Differenzierung aufgefasst. Sie bezeichnen API–Methoden, wonach sich beide Messgrößen auf Methodendeklarationen in Programmierschnittstellen beziehen. Die gegensätzliche Richtung dieser Beziehungen zeigen durch den entsprechenden Indikator Methodendeklarationen in Projekten an.

Overriden method

```
% over_method(Pid,Aid,Classifier,Name,Arguments,Return,Modifiers)
% m in c1 and c2, c1 extends c2, c2 not abstract
over_method(Pid,Aid,C,N,As,R,Ms) :-
    method(Aid,C,N,As,R,Ms),
    full_name(P_a, N_a, C),
    % not from an abstract class
    class(Aid,P_a,N_a,_,_,Ms_a),
    \+ member('abstract', Ms_a),
    % same method by name and arguments
    method(Pid,C_p,N,As,_,_),
    full_name(P_p ,N_p, C_p),
    % project class extends api class
    class(Pid,P_p,N_p,C,_,_).
```

Implemented method

```
% impl_method(Pid,Aid,Classifier,Name,Arguments,Return,Modifiers)
% m in c1 and c2, c1 extends c2, c2 abstract
impl_method(Pid,Aid,C,N,As,R,Ms) :-
    method(Aid,C,N,As,R,Ms),
    full_name(P_a, N_a, C),
    % from an abstract class
    class(Aid,P_a,N_a,_,_,Ms_a),
    member('abstract', Ms_a),
    % same method by name and arguments
    method(Pid,C_p,N,As,_,_),
    full_name(P_p, N_p ,C_p),
    % project class extends api class
    class(Pid,P_p,N_p,C,_,_).
% m in c and i, c implements i
impl_method(Pid,Aid,C,N,As,R,Ms) :-
    method(Aid,C,N,As,R,Ms),
    full_name(P_a, N_a, C),
```

```
    interface(Aid,P_a,N_a,_,_),
    % same method by name and arguments
    method(Pid,C_p,N,As,_,_),
    full_name(P_p, N_p, C_p),
    % project class implements api class
    class(Pid,P_p,N_p,_,Is,_),
    member(C, Is).
```

Derived method

```
% drvd_method(Pid,Aid,Classifier,Name,Arguments,Return,Modifiers)
% m in c1 and c2, c1 extends c2
drvd_method(Pid,Aid,C,N,As,R,Ms) :-
    method(Pid,C,N,As,R,Ms),
    full_name(P_p, N_p, C),
    class(Pid,P_p,N_p,S,_,_),
    % same method by name and arguments
    method(Aid,S,N,As,_,_).
% m in c and i, c implements i
drvd_method(Pid,Aid,C,N,As,R,Ms) :-
    method(Pid,C,N,As,R,Ms),
    full_name(P_p, N_p, C),
    class(Pid,P_p,N_,_,Is,Ms),
    % same method by name and arguments
    method(Aid,C_a,N,As,_,_),
    member(C_a, Is).
```

4.1.3 Verwendung von abgeleiteten Elementen

Ausgehend von den gewonnenen Informationen über Code–Elemente, die aus APIs abgeleitet sind, werden Verwendungen dieser Komponenten untersucht. Beide verwendeten Indikatoren verweisen dabei entsprechend auf `methodCall`–Prädikate in Projekten.

Frameworky call

```
% framewky_call(Pid,Aid,Callee,Arguments)
% method call of a derived method
framewky_call(Pid,Aid,Callee,A) :-
    methodCall(Pid,_,_,Callee,A),
    full_name(C, N, Callee),
    drvd_method(Pid,Aid,C,N,A,_,_).
```

Frameworky use

```
% framewky_use(Pid,Aid,Callee)
% constructor call of a derived class
framewky_use(Pid,Aid,Callee) :-
    methodCall(Pid,_,_,Callee,_),
    full_name(Classifier, '<init>', Callee),
    full_name(P, N, Classifier),
    (   standard_drvd_class(Pid,Aid,P,N,_,_,_)
    ;   framewky_drvd_class(Pid,Aid,P,N,_,_,_)
    ).
```

4.1.4 Referenzen von API–Typen

Die diversen Referenzen von Typen im Projektcode sind ebenfalls Ziel der Analyse. Die Verweise durch

- Felder
- Aufrufe statischer Methoden
- Argumente von Methoden
- Rückgabewerte von Methoden
- lokale Variablen

werden dabei berücksichtigt und als Prolog–Fakten mit dem entsprechenden vollständig qualifizierten Typnamen als Zeiger codiert.

Reference by field

```
% ref_field(Pid,Aid,Type)
% api type as field
ref_field(Pid,Aid,Type) :-
    % field in project
    field(Pid,_,Type,_),
    full_name(P, N, Type),
    % any api type
    (   class(Aid,P,N,_,_,_)
    ;   interface(Aid,P,N,_,_)
    ).
```

Reference by call of static method

```
% ref_static_method(Pid,Aid,Type)
% called static method of an api type
ref_static_method(Pid,Aid,Type) :-
    method(Aid,Type,N,As,_,Ms),
    member('static', Ms),
    full_name(Type, N, Callee),
    methodCall(Pid,_,_,Callee,As).
```

Reference by argument

```
% ref_argument(Pid,Aid,Type)
% api type as argument
ref_argument(Pid,Aid,Type) :-
    method(Pid,C,N,As,_,_,_),
    member(Type, As),
    full_name(P, N, Type),
    % any api type
    (   class(Aid,P,N,_,_,_)
    ;   interface(Aid,P,N,_,_)
    ).
```

Reference by return type

```
% ref_return_type(Pid,Aid,Type)
% api type as return type
ref_return_type(Pid,Aid,Type) :-
    method(Pid,C,N,_,Type,_,_),
    full_name(P, N, Type),
    % any api type
    (   class(Aid,P,N,_,_,_)
    ;   interface(Aid,P,N,_,_)
    ).
```

Reference by local variable

```
% ref_localVariable(Pid,Aid,Type)
% api type as local variable
ref_localVariable(Pid,Aid,Type) :-
    % local variable in project
    localVariable(Pid,_,Type,_),
    full_name(P, N, Type),
```

```
    % any api type
    (   class(Aid,P,N,_,_,_)
    ;   interface(Aid,P,N,_,_)
    ).
```

4.2 Aufbau und Struktur von APIs

4.2.1 Interne abstrakte Basistypen

Interne Basistypen beschreiben die API hinsichtlich der Fragestellung, wie der hierarchische Aufbau innerhalb der Programmierschnittstelle strukturiert ist. Dementsprechend verweisen die Messgrößen für interne abstrakte Basistypen auf entsprechende API–Typen, die innerhalb der Programmierschnittstelle in irgendeiner Form erweitert oder implementiert werden.

Intern base abstract class

```
% ibase_abstr_class(Aid,Package,Name,Supertypes,Modifiers)
% extended abstract class
ibase_abstr_class(Aid,P,N,Ss,Ms) :-
    [S|Is] = Ss,
    class(Aid,P,N,S,Is,Ms),
    % only abstract base types
    member('abstract', Ms),
    full_name(P, N, Full),
    % extended by some class
    class(Aid,_,_,Full,_,_).
```

Intern base interface

```
% ibase_interface(Aid,Package,Name,Supertypes,Modifiers)
% implemented interface
ibase_interface(Aid,P,N,Ss,Ms) :-
    interface(Aid,P,N,Ss,Ms),
    full_name(P, N, Full),
    % implemented by some class
    class(Aid,_,_,_,Is,_),
    member(Full, Is).
% extended interface
ibase_interface(Aid,P,N,Ss,Ms) :-
    interface(Aid,P,N,Ss,Ms),
```

```
    full_name(P, N, Full),
    % extended by some interface
    interface(Aid,_,_,E_,_),
    member(Full, Es).
```

4.2.2 Non–subtyped Klassen und Interfaces

In ähnlicher Form werden die entgegengesetzten abstrakten Typen beschrieben. Diese Typen werden bei dieser Vorgehensweise unterteilt in Komponenten ohne echte Implementation innerhalb der API und in Elemente, die ihre Funktionalität ausschließlich durch nicht–öffentliche Klassen erhalten.

By all types unextended abstract class

```
% by_all_unext_abstr_class(Aid,Package,Name,Supertypes,Modifiers)
% no implementation
by_all_unext_abstr_class(Aid,P,N,Ss,Ms) :-
    [S|Is] = Ss,
    class(Aid,P,N,S,Is,Ms),
    member('abstract', Ms),
    full_name(P, N, Full),
    (   % no subtype
        \+ class(Aid,_,_,Full,_,_)
    ;   % abstract subtype
        class(Aid,_,_,Full,_,Ms_a),
        member('abstract', Ms_a)
    ).
```

By public types unextended abstract class

```
% by_pub_unext_abstr_class(Aid,Package,Name,Supertypes,Modifiers)
% non-public implementation
by_pub_unext_abstr_class(Aid,P,N,Ss,Ms) :-
    [S|Is] = Ss,
    class(Aid,P,N,S,Is,Ms),
    member('abstract', Ms),
    full_name(P, N, Full),
    % extended by non-public class
    class(Aid,_,_,Full,_,Ms_a),
    \+ member('public', Ms_a).
```

By all types unimplemented interface

```
% by_all_unimpl_interface(Aid,Package,Name,Supertypes,Modifiers)
% no implementation
by_all_unimpl_interface(Aid,P,N,Ss,Ms) :-
    interface(Aid,P,N,Ss,Ms),
    full_name(P, N, Full),
    (   % no subtype
        \+ class(Aid,_,_,Full,_,_)
    ;   % abstract subtype
        class(Aid,_,_,Full,_,Ms_a),
        member('abstract', Ms_a)
    ).
```

By public types unimplemented interface

```
% by_pub_unimpl_interface(Aid,Package,Name,Supertypes,Modifiers)
% non-public implementation
by_pub_unimpl_interface(Aid,P,N,Ss,Ms) :-
    interface(Aid,P,N,Ss,Ms),
    full_name(P, N, Full),
    % implemented by non-public class
    class(Aid,_,_,_,Is,Ms_a),
    member(Full, Is),
    \+ member('public', Ms_a).
```

4.2.3 Abgeschlossene Typen

Abgeschlossene bzw. normale konkrete Klassen beziehen sich jeweils auf API–Typen, die entweder als `final` deklariert sind, oder von einem anderen API–Element abgeleitet sind. In letzterem Fall wird zwischen öffentlichen und nicht–öffentlichen Klassen unterschieden, um einen Gegenwert zu den abstrakten Typen zu erhalten, die nur durch öffentliche Komponenten eine Implementation erhalten.

Sealed class

```
% sealed_class(Aid,Package,Name,Supertypes,Modifiers)
% final class
sealed_class(Aid,P,N,Ss,Ms) :-
    [S|Is] = Ss,
    class(Aid,P,N,S,Is,Ms),
    member('final', Ms).
```

Non–sealed derived public class

```
% nsealed_drvd_pub_class(Aid,Package,Name,Supertypes,Modifiers)
% public, non-final, extending/implementing class
nsealed_drvd_pub_class(Aid,P,N,Ss,Ms) :-
    [S|Is] = Ss,
    class(Aid,P,N,S,Is,Ms),
    \+ member('final', Ms),
    member('public', Ms),
    class(Aid,P_a,N_a,_,_,_),
    % extending or implementing
    full_name(P_a, N_a, Full),
    (   Full = S
    ;   member(Full, Is)
    ).
```

Non–sealed derived non-public class

```
% nsealed_drvd_npub_class(Aid,Package,Name,Supertypes,Modifiers)
% non-public, non-final, extending/implementing class
nsealed_drvd_npub_class(Aid,P,N,Ss,Ms) :-
    [S|Is] = Ss,
    class(Aid,P,N,S,Is,Ms),
    \+ member('final', Ms),
    \+ member('public', Ms),
    class(Aid,P_a,N_a,_,_,_),
    % extending or implementing
    full_name(P_a, N_a, Full),
    (   Full = S
    ;   member(Full, Is)
    ).
```

Kapitel 5

Messung und Bewertung

Die im folgenden Kapitel vorgestellten Resultate sind jeweils mit einer Hinweisbox versehen, welche die entsprechende Grafik oder Tabelle unterstützend mit Informationen versieht.

Erkärung zur Hinweisbox:
data
Erläuterung der im Diagramm oder in der Tabelle vorkommenden Daten
sorted by
Spezifizierung der Reihenfolge der Daten
filtered by
Angabe, welche Filter auf die gesamte Datenbasis angewendet wurden

5.1 Nachweis von Framework–Verwendung

Grundlage aller Untersuchungen bildet der Nachweis, ob eine API als Framework Verwendung findet. Die Tabelle **??** stellt diesbezüglich eine Übersicht dar, in welcher zum einen die Verbreitung der Verwendung in Projekten, zum anderen die absolute Anzahl an Framework–Elementen deutlich wird. Es ist zu erkennen, dass die „framework–hafte“ Verwendung von APIs in den Projekten gering ausfällt. Die einzigen Ausnahmen bilden die GUI–APIs `AWT` und `Swing`, die XML–APIs `SAX` und `Xerces`, sowie die `Collection`–API.

Darüber hinaus ist festzustellen, dass mit der Programmierschnittstelle `Collection` eine Art Grundkomponente existiert, die in fast allen Projekten Verwendung findet.

API	# Projects			# Methods		# Distinct methods		# Types		# Distinct types	
	interf.	classes	any	impl.	over.	impl.	over.	interf.	classes	interf.	classes
Collection	29	23	30	795	216	84	82	480	123	9	15
AWT	20	18	21	3182	771	56	73	2500	606	21	25
Swing	16	19	19	1684	1561	142	363	617	1561	43	108
SAX	8	14	17	160	286	30	31	37	63	7	5
xerces	8	14	17	162	286	32	31	38	63	8	5
junit	1	6	7	1	96	1	3	1	531	1	3
Reflection	6	0	6	8	0	5	0	7	0	3	0
servlet	3	5	5	407	114	139	34	32	60	17	13
RMI	4	4	4	0	0	0	0	8	8	1	3
ant	1	4	4	692	608	190	226	232	534	64	112

Tabelle 5.1: Nachweis von Framework–Verwendung

Hinweise zur Tabelle **??**:

data
Projects Verbreitung der Verwendung einer als Framework
Methods Anzahl an Basis–Methoden
Distinct methods Anzahl an unterschiedlichen Basis–Methoden
Types Anzahl an Basis–Typen
Distinct types Anzahl an unterschiedlichen Basis–Typen

sorted by
any Anzahl an Projekten, die die API in irgendeiner Form als Framework nutzen

filtered by
any APIs mit den zehn höchsten Werte

Somit sind solche Container–Typen und dazugehörige Hilfstechnologien eine wiederkehrende Verwendung der Laufzeitumgebung von Java.

Interessanterweise weisen die Zahlen von allen und unterschiedlichen API–Supertypen deutliche Diskrepanzen auf. Der erweiternde und implementierende Umgang mit Java–APIs konzentriert sich demzufolge auf bestimmte Kernfunktionalitäten, welche wiederholt Verwendung finden. Verdeutlicht wird dies durch die Tag–Cloud aus Abbildung **??**, indem stark genutzte Features hervorgehoben werden. Die zentralen Begriffe verweisen dabei allesamt auf GUI–Elemente wie die Nutzung von Listener–Schnittstellen bzw. deren Adapter–Klassen, wobei hier explizit die Verwendung des Interface `java.awt.event.ActionListener` herauszustellen ist. Betrachtet man die Werte der API `AWT` unter diesem Hintergrund genauer, ist zu erkennen, wie sich die Nutzung dieser API in den Projekten primär auf das eine Interface konzentriert.

Die Tabelle weist aber auch daraufhin, dass Erweiterungen und Implementierungen generell Bestandteil im Gebrauch von verschiedenen APIs und Anwendungsgebieten ist, da in diesem Ausschnitt aus der Menge aller APIs mit Datenstrukturen,

Abbildung 5.1: Erweiterte und implementierte Features

> Hinweise zur Abbildung **??**:
>
> **data**
> ***Feature*** Aufteilung der *camelCase*–Notation der Namen von erweiterten und implementierten Typen und Gewichtung nach Häufigkeit
>
> **sorted by**
> -
>
> **filtered by**
> ***Feature*** maximal 150 Wörter

GUI, XML, Testen etc. mehrere Domänen vertreten sind, wobei quantitativ GUI und XML dominieren.

Betracht man davon abweichend die relative Anzahl an unterschiedlichen Erweiterungen und Implementierung im Bezug zur Gesamtmenge an möglichen Supertypen (vgl. Abbildung **??**), verschiebt sich das bisher gezeichnete Bild etwas. So finden sich die GUI–APIs `AWT` und `Swing` zwar wieder, jedoch ist deren *coverage*–Wert im Vergleich zur `Servlet`–API deutlich geringer. Dies ist insofern nicht verwunderlich, da sich, wie bereits erwähnt, die Verwendung von `AWT` und `Swing` stark auf einzelne Features fokussiert, während diese APIs darüber hinaus eine Vielzahl an anderen GUI–Komponenten bereitstellt wie zum Beispiel zur Kontrolle (Buttons, Menüs), Anzeige (Labels, Icons), Textverarbeitung (Textfelder, Tabellen) oder Anordnung (Layout–Manager). Auf der anderen Seite existiert mit `Jetty` ein Projekt, welches einen in Java geschriebenen Servlet/JSP–Container und Webserver darstellt und daher die Möglichkeiten der `Servlet`–API intensiv ausnutzt. Im Vergleich zu den anderen Programmierschnittstellen ist eine deutliche Ausnutzung der `SAX`–API ebenfalls gegeben. Der innere Aufbau bedingt diese Abdeckung, da das *XML Information Set*[1] für sich, also die Menge an unterschiedlichen Informationsträgern eines XML-Dokuments, über neun

[1] W3C Standard, vgl. http://www.w3.org/TR/xml-infoset/

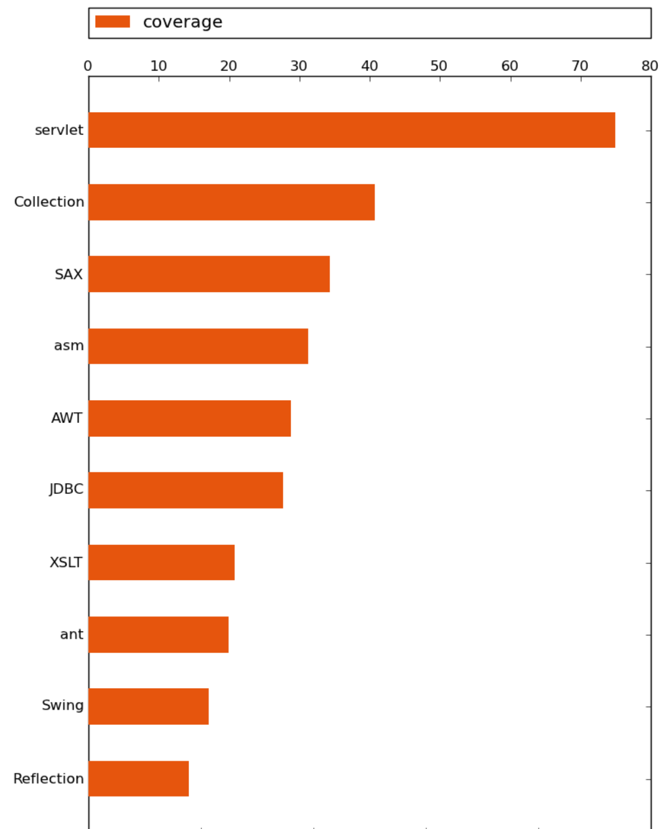

Abbildung 5.2: Coverage–Metrik

Hinweise zur Abbildung **??**:

data
Subtyping coverage in Prozent

sorted by
Subtyping coverage Absteigend nach den Werten der Metrik

filtered by
Public types APIs mit mindestens 10 öffentliche Typen, 10 höchsten Metrik–Werte

verschiedene Schnittstellen modelliert ist[2]. Hinzukommen einige vorgegebene Implementationen, die ebenfalls als Vorlage Verwendung finden und einen Teil der oben herausgestellten Informationsträger–Schnittstellen implementieren. Eine umfassende XML–Anwendung muss daher einen Großteil des Objektmodells erweitern oder implementieren.

5.2 Struktureller Aufbau von APIs

Die Betrachtung derart spezieller Typen legt eine Analyse der API nahe, die gezielt deren Typen in ihre verschiedenen Funktionalitäten auffächert, um so die Intention der einzelnen Typen und damit die Konzeption der gesamten API herauszustellen. Die Abbildung **??** leistet dies, indem sie die Typen einer API in vier Hauptgruppen einteilt:

1. Abstrakte Typen, die keine konkrete abgeleitete Klasse in der API besitzen
2. Abstrakte Typen, die über solch eine Subklasse verfügen
3. Konkrete Klassen, die erweitert werden können
4. Abgeschlossene Klassen (*sealed classes*)

Bemerkenswert sind zunächst zwei Aspekte:

- Die APIs `DOM` und `ejb3 persistence` bestehen im wesentlichen nur aus Schnittstellen der Kategorie (1).
- Es sind fünf XML–APIs vertreten .

Versucht man den ersten dieser beiden Punkte nachzuvollziehen stellt man fest, dass es sich bei der `DOM`–API respektive dem `org.w3c.dom.*` um die Sprachanbindung des entsprechenden W3C–Standards[3] handelt. Dieser beinhaltet keine konkrete Implementierung, sondern ist lediglich eine auf Schnittstellen basierte Spezifikation, die zusätzlich über ein paar Klassen für Ausnahmen und Fehler verfügt. Analoge Erkenntnisse liefert auch eine Betrachtung der `StAX`-API, die darüber hinaus jedoch zum einen Factory–Klassen zur Erzeugung von XML–Events und Behandlung von Input und Output beinhaltet, zum anderen aber auch „leere“ Subklassen von den Schnittstellen der Parser, die keine andere Funktionalität beinhalten als das Weiterleiten der

[2] s. [**?**, Tabelle T1]
[3] vgl. http://www.w3.org/DOM/

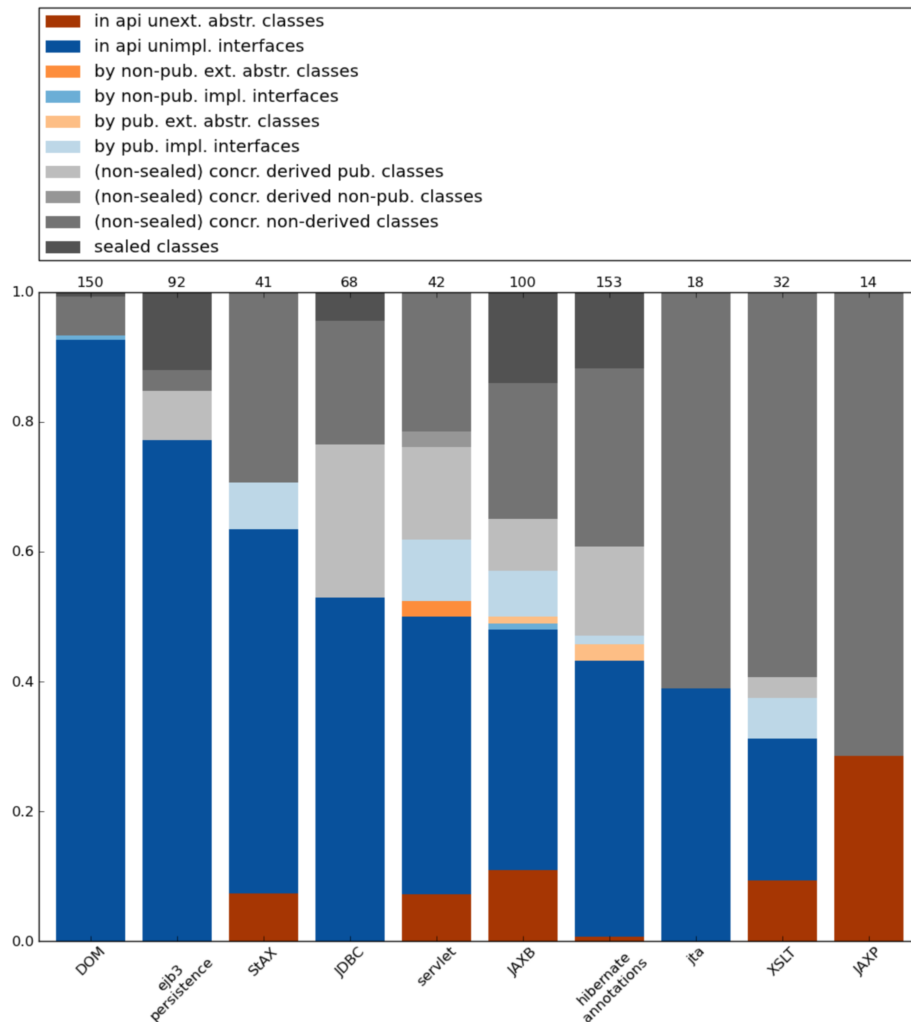

Abbildung 5.3: Strukturelle Einordnung der API-Typen

Hinweise zur Abbildung **??**:

data
API types Aufschlüsselung der verschiedenen API–Typen, Bezifferung der Gesamtzahl an API–Typen in der oberen x–Achse

sorted by
non suptyped Absteigend nach der Anzahl an Typen, die innerhalb der API selbst nicht erweitert oder implementiert sind

filtered by
non suptyped APIs mit den zehn höchsten Sortierwerten

Abbildung 5.4: Features ohne konkreten Subtyp

Hinweise zur Abbildung **??**:

data
Feature Aufteilung der *camelCase*–Notation der Namen von abstrakten Typen ohne konkrete Subklassen

sorted by
-

filtered by
Feature maximal 150 Wörter

Methodenaufrufe an die Basisschnittstellen. Die API `ejb3 persistence` hingegen stellt diverse Annotation–Typen bereit, welche in ihre Beschaffenheit als spezielle Schnittstellen wie Interface–Deklarationen behandelt werden. Die Programmierschnittstelle stellt auf diese Weise ähnlich wie die Annotation–API von `hibernate` eine Methode zur objektrelationalen Abbildung bereit, die intern dementsprechend ungenutzt bleibt.

Betrachtet man die API–Features, die durch Typen der erste Kategorie bereitgestellt werden isoliert (s. Abbildung **??**), lässt sich eine große Vielfalt erkennen. Dabei steht mit `Listener` und `Adapter` wiederum die Domäne GUI im Vordergrund, wobei das Anwendungsgebiet XML ebenfalls zu erkennen ist. Das Feature `Factory` beispielsweise verweist im Wesentlichen auf XML, da Factory–Klassen typisch bei der Verarbeitung von XML–Dokument sind und Klassen wie `DocumentBuilderFactory` und `SAXParserFactory` oder auch die bereits erwähnten Klassen der `StAX`–API allesamt als abstrakt deklariert sind.

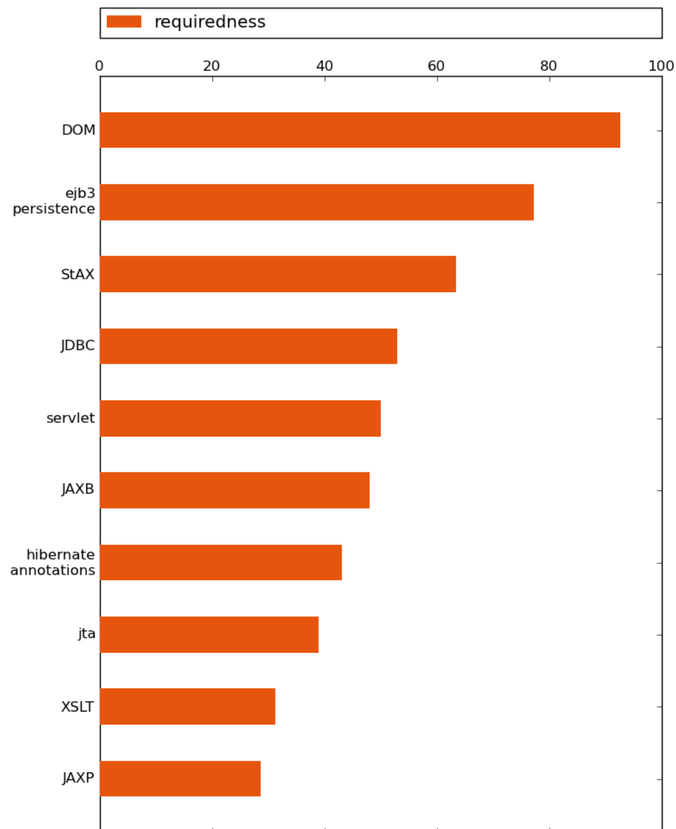

Abbildung 5.5: Requiredness–Metrik

Hinweise zur Abbildung **??**:

data
Subtyping requiredness in Prozent

sorted by
Subtyping requiredness Absteigend nach den Werten

filtered by
Subtyping requiredness APIs mit den zehn höchsten Werte, 10 höchsten Metrik–Werte

Die *requiredness*–Metrik in Abbildung **??** greift die Grafik **??** nochmals auf und fokussiert jeweils die beiden unteren Stapel der einzelnen Balken, also die Anzahl an abstrakten Typen ohne Subtyp innerhalb der API. Es wird daher deutlich, dass im Wesentlichen vier Programmierschnittstellen erweitert oder implementiert werden müssen, um konkret in einer Anwendung Nutzung zu erfahren. Dies triff in besonderem Maße auf `DOM` und `StAX`, sowie mit Abstrichen auch auf `JAXB` und `servlet` zu. In dieser Hinsicht sind die auf Annotations basierten APIs nicht zu berücksichtigen, da die Verwendung eines Annotation–Typs nicht mit einer Implementierung gleichzusetzen ist. Ebenso ist die Programmierschnittstelle `JDBC` zu ignorieren, da diese Datenbank–API als Vereinheitlichung zwischen den verschiedenen Herstellern anzusehen ist und somit diesen Unternehmen die Implementierung obliegt.

Die gegensätzliche Darstellung, also die *inapplicability*–Metrik, ist in Abbildung **??** visualisiert. Es ist festzuhalten, dass diese Eigenschaft vor allem in sicherheitsrelevanten Bereichen von Programmierschnittstellen Verwendung findet, wie es in [**?**] als Anwendungsfall beschrieben ist. Betroffene Gebiete sind dabei u. a. Management des Cache im Fall der APIs `ehcache` und `oro`, fest vorgegebene Spezifikationen des Objektmodells, wie es in `commons net` durch die Serverkommunikation gegeben ist und Abbildungen von Objekten auf externe Darstellungsformen (`serializer`–API). Die Programmierschnittstelle `Reflection` stellt, um mit Feldern, Methoden und Konstruktoren von geladenen Typen zu arbeiten, eine Umsetzung für eben diese Funktionalität bereit, die bewusst abgeschlossen ist, damit dieser sensible Bereich geschützt bzw. der Zugriff und der Informationsgehalt reglementiert ist.

Einen interessanten Aspekt erörtert die Unterteilung der zweiten Kategorie, indem unterschieden wird zwischen abstrakten Typen, die durch öffentliche API-Typen implementiert werden, und solche, die nicht–öffentliche Subklassen haben. Auf diese Weise ist es möglich API–interne, also aus Sicht der Anwendung versteckte Implementierungen zu finden, die beispielsweise durch Factory–Methoden genutzt werden können. Insbesondere der Entwurf, also das Fehlen einer konkreten Implementation, der XML–APIs `DOM` und `StAX` kann die Annahme hinsichtlich der Existenz solcher Subklassen bestärken. Diese Vermutung ist jedoch zurückzuweisen, da die Abbildung **??** eine derartige Annahme eindeutig widerlegt.

5.2.1 Provider–Implementationen

Diese Erkenntnis steht im starken Widerspruch zur praktischen Nutzung von `DOM` bei der XML–Verarbeitung, wie es im Codebeispiel **??** dargestellt ist. Diese Vorgehens-

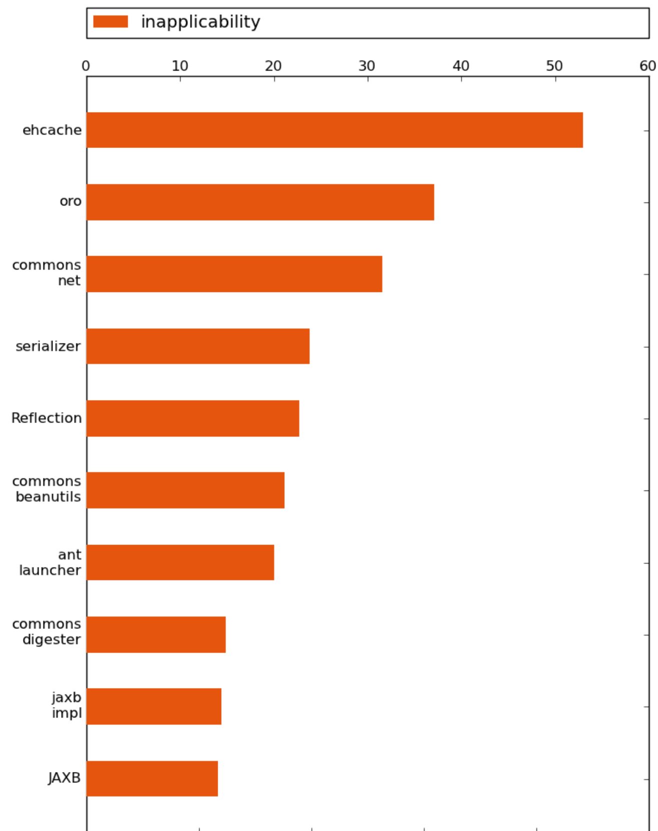

Abbildung 5.6: Inapplicability–Metrik

Hinweise zur Abbildung **??**:

data
Subtyping inapplicability in Prozent

sorted by
Subtyping inapplicability Absteigend nach den Werten

filtered by
Subtyping inapplicability APIs mit den zehn höchsten Werte

```
1 DocumentBuilderFactory docFactory = DocumentBuilderFactory.
     newInstance();
2 DocumentBuilder docBuilder = docFactory.newDocumentBuilder();
3
4 Document doc = docBuilder.newDocument();
5 Element rootElement = doc.createElement("test");
```

Listing 5.1: typischer Umgang mit der `DOM`-API

weise nutzt dabei DOM direkt, ohne im Vorfeld eine konkrete Implementierung zu schaffen. Es stellt sich demnach die Frage, welche Implementation genutzt wird, da die Programmierschnittstelle selbst keine enthält. Zur Klärung dieses Sachverhalts ist zunächst darauf hinzuweisen, dass der gezeigte Ansatz im Umgang mit `DOM` diese API nicht unmittelbar verwendet, sondern dass die Erzeugung des `Document`–Objekts über die Java API für XML Processing (`JAXP`) erfolgt und erst durch das Anlegen des Objekts vom Typ `Element` die `DOM`–API Verwendung findet. Dabei zielt die tatsächliche Erzeugung der `Document`–Instanz, also der Aufruf der `newInstance`–Methode, auf eine abstrakte Methode ab, woraus folgt, dass die `Factory`–Klasse die `DOM`–Implementation bestimmt. Zur Vorgehensweise dazu heißt es in der Java–Dokumentation[4]:

> This method uses the following ordered lookup procedure to determine the `DocumentBuilderFactory` implementation class to load:
>
> - Use the `javax.xml.parsers.DocumentBuilderFactory` system property.
> - Use the properties file "lib / jaxp.properties" in the JRE directory. This configuration file is in standard `java.util.Properties` format and contains the fully qualified name of the implementation class with the key being the system property defined above. The jaxp.-properties file is read only once by the JAXP implementation and it's values are then cached for future use. If the file does not exist when the first attempt is made to read from it, no further attempts are made to check for its existence. It is not possible to change the value of any property in jaxp.properties after it has been read for the first time.
> - Use the Services API (as detailed in the JAR specification), if available, to determine the classname. The Services API will look for a classname in the file `META-INF/services/` (...) in jars available to the runtime.

[4] s. [?, Methode `newInstance()`]

- Platform default `DocumentBuilderFactory` instance.

Als weitere Strategie ist dabei sogar ein Ersatzwert fest im Sourcecode implementiert, falls der zitierte Ansatz keine entsprechende Implementation findet (s. Code-Ausschnitt **??**). Es existieren also externe Implementationen von APIs, die als Provider für die spezifischen Funktionalitäten fungieren. Im beschrieben Fall wird auf die `DOM`–Implementation der `Xerces`–API zurückgegriffen, welche im Package `com.sun.org.apache.xerces.internal.dom.*` mitgeliefert wird. Ebenso beinhaltet die Java–Laufzeitumgebung Packages (`com.sun.xml.internal.stream.*`), welche die `StAX`–API mit konkreten Implementationen versieht.

```
1  public static DocumentBuilderFactory newInstance() {
2    try {
3      return (DocumentBuilderFactory) FactoryFinder.find(
4      /* The default property name according to the JAXP spec */
5      "javax.xml.parsers.DocumentBuilderFactory",
6      /* The fallback implementation class name */
7      "com.sun.org.apache.xerces.internal.jaxp.
          DocumentBuilderFactoryImpl");
8    catch (FactoryFinder.ConfigurationError e) {
9       throw new FactoryConfigurationError(e.getException(),
10     e.getMessage());
11   }
12 }
```

Listing 5.2: *newInstance*-Methode der DocumentBuilderFactory Klasse [**?**]

Um diesen Umstand in der Untersuchung zu berücksichtigen, sind wichtige Ergänzungen durchzuführen. So muss ein kompletter Analyseschritt hinzugefügt werden, in welchem überprüft wird, ob von API–Typen ohne konkreten Subtyp innerhalb der Programmierschnittstelle selbst irgendwo anders konkrete Klassen abgeleitet werden. Es gilt demnach versteckte Implementationen aufzuspüren, welche Provider zu bestimmten Programmierschnittstellen darstellen. Dies erfolgt zum einen in der Laufzeitumgebung, also im Speziellen in der `rt.jar`, zum anderen im API–Pool. Somit ist ein neuer *anaylzer* notwendig, welcher die Ergebnismenge um zwei neue dynamische Prädikate erweitert und die Ergebnismenge des *API anaylzer* modifiziert.[5]

Die auf diese Weise neu entstehende Abbildung der API–Typen zeigt deutliche Veränderungen auf (vgl. Abbildung **??**). So nimmt der Anteil an Typen, die nicht erweitert oder implementiert sind, erheblich ab, woraus zu schließen ist, dass im Endeffekt alle APIs mit einer Grundimplementation versehen sind, sei es durch die Pro-

[5] Entsprechende Darstellungen sind im Anhang in Abschnitt **??** zu finden

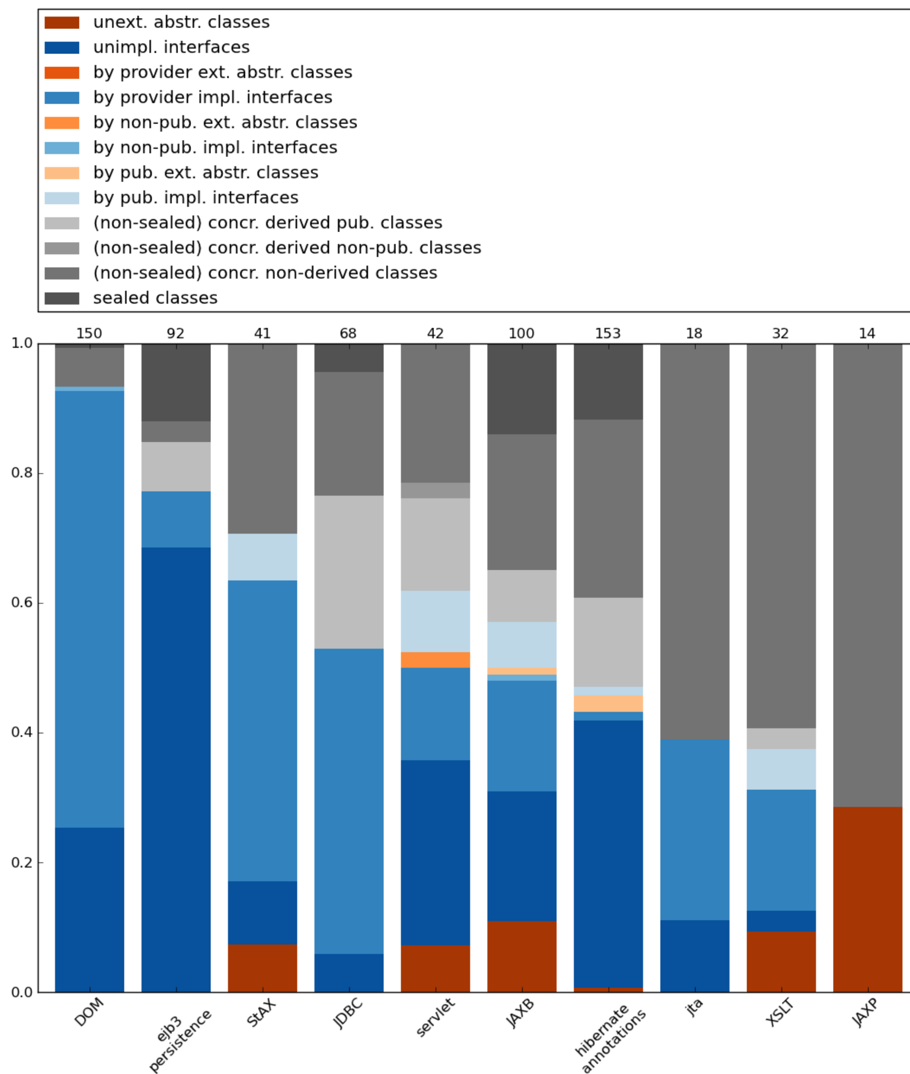

Abbildung 5.7: Strukturelle Einordnung der API-Typen mit Provider–Analyse

Hinweise zur Abbildung **??**: (Analog zu Abbildung **??**)

data
API types Aufschlüsselung der verschiedenen API–Typen, Bezifferung der Gesamtzahl an API–Typen in der oberen x–Achse

sorted by
non suptyped Absteigend nach der Anzahl an Typen, die innerhalb der API selbst nicht erweitert oder implementiert sind

filtered by
non suptyped APIs mit den zehn höchsten Sortierwerten

grammierschnittstelle selbst oder durch einen Provider (die auf Annotations basierten APIs sind von dieser Betrachtung wiederum auszuschließen). Somit ist an dieser Stelle festzuhalten, dass die *subtyping requiredness* alleine kein hinreichendes Kriterium für eine Verwendung als Framework ist, da sich dieser Indikator auf Grund der Existenz von Provider–Implementierungen nicht im Anwendungscode widerspiegeln muss.

5.3 Ausnutzung der API–Strukturen

Diese Erkenntnis reflektiert die Grafik **??** unter einem andere Gesichtspunkt, indem sie alle Supertypen einer Programmierschnittstelle in standardmäßige und „framework–hafte“ Typen auffächert. Dabei ist festzustellen, dass zwar „framework–hafte“ Basistypen existieren und demnach in den Projekten genutzt werden, jedoch leitet sich der Großteil an Erweiterungen und Implementierungen aus normalen Typen ab. Ein Ausnahme bildet einzig die `JDBC`–API, da mit den Projekten `axion` und `hsqldb` zwei relationale Datenbanksysteme im Corpus enthalten sind, die von den abstrakt gehaltenen Typen im Sinne eines Herstellersystems Gebrauch machen.

Interessanter als dieses Verhältnis ist jedoch eine Untersuchung mit der Zielsetzung, ob die Typen ohne konkrete Subklassen genutzt werden bzw. der Hauptanteil überhaupt nicht erweitert oder implementiert wird. Die Abbildung **??** zeigt auf, dass von dieser Möglichkeit nur wenig Gebrauch gemacht wird, so dass eigentlich nur bei den Programmierschnittstellen `servlet` und `AWT` die Konzeption ausgenutzt ist. Dies ist insofern nicht verwunderlich, da im Fall der `servlet`–API diese Ausnutzung durch das Projekt `Jetty` bedingt ist und GUI–Anwendung wie `iReport`, `JEdit`, `MegaMek` usw. den abstrakten Bereich der Ereignisbehandlung intensiv verwenden.

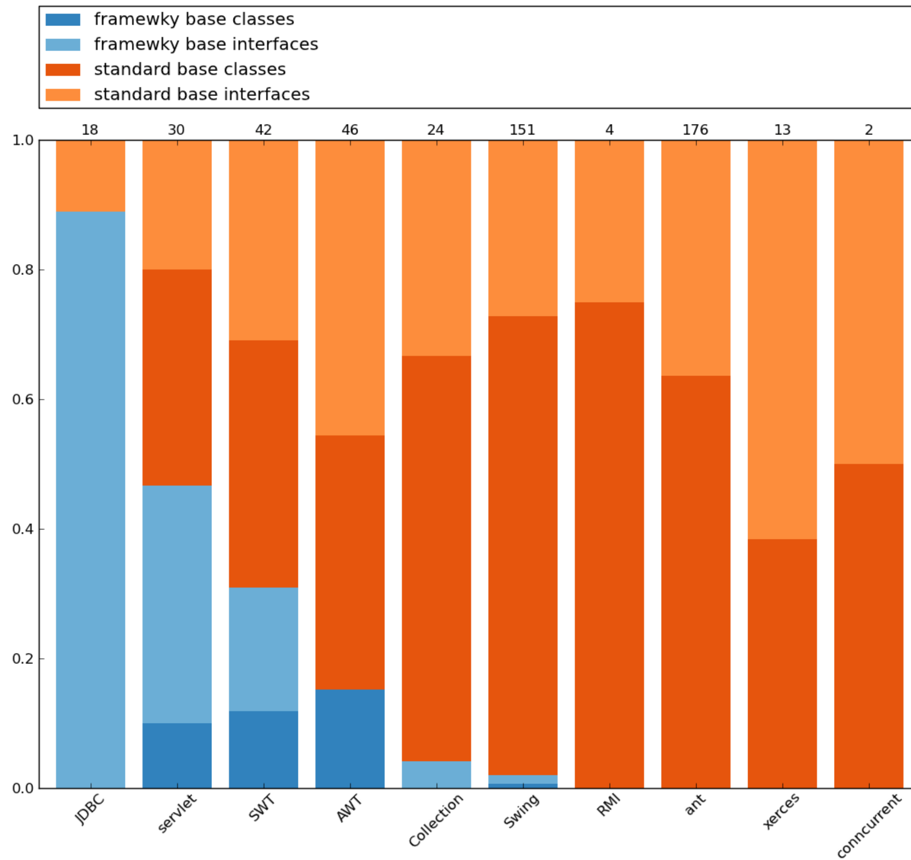

Abbildung 5.8: Strukturelle Einordnung der API–Basistypen

Hinweise zur Abbildung **??**:

data
base types Verhältnis zwischen *frameworky* und *standard* Supertypen, Bezifferung der Gesamtzahl an Superklassen durch den Wert der oberen x–Achse

sorted by
framewky base types Absteigend nach der Anzahl an *frameworky* Supertypen

filtered by
base types APIs mit mindestens zehn Supertypen

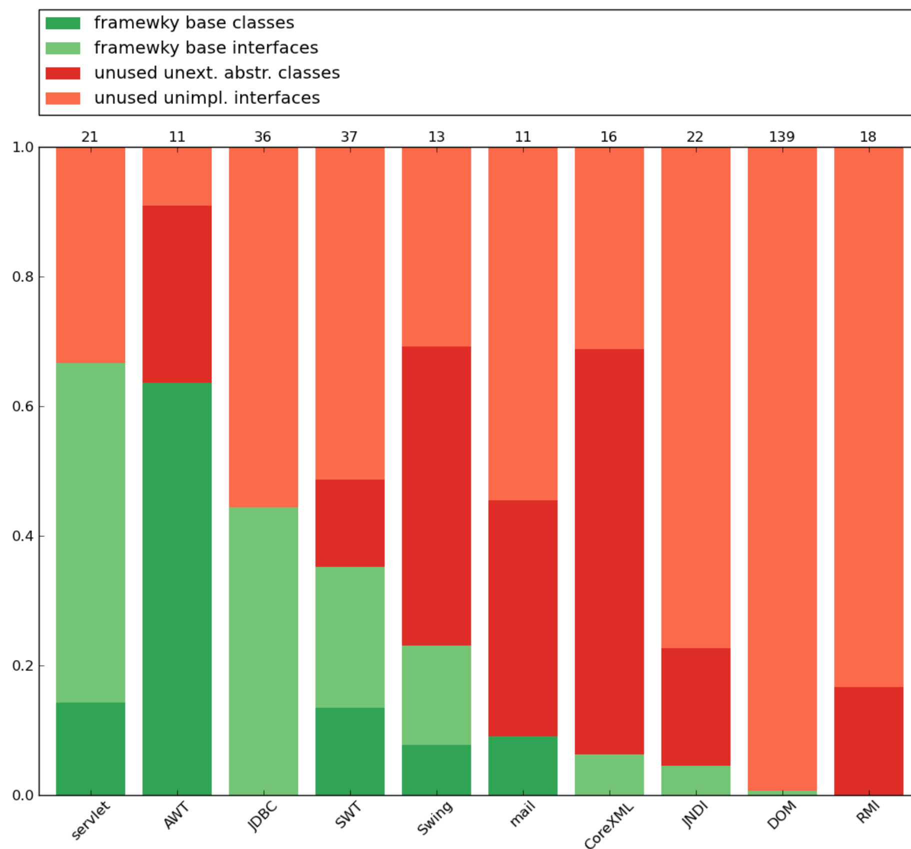

Abbildung 5.9: Ausnutzung der API–Strukturen beim Subtyping

Hinweise zur Abbildung **??**:

data
non subtyped Anteil an API–Typen, die innerhalb dieser keinen konkreten Subtyp besitzen, durch ein beliebiges Projekt eine Implementation erhalten, Bezifferung der Gesamtzahl an solchen Typen durch den Wert der oberen x–Achse

sorted by
framewky base types Absteigend nach der Anzahl an *frameworky* Supertypen

filtered by
non subtyped APIs mit mindestens zehn entsprechenden Typen

Kapitel 6

Verwandte Arbeiten

Diese Ausarbeitung berührt durch die Analyse der *Frameworkiness* von APIs im Wesentlichen zwei Forschungsgebiete, namentlich die Untersuchung über die Verwendung von APIs und objektorientierte Frameworks.

API–usage analysis Im Bereich der *API–usage analysis* existieren bereits eine Vielzahl an Untersuchungen mit unterschiedlichen Intentionen. Einen großen Bereich bilden in diesem Zusammenhang dabei Data–Mining–Ansätze wie in [?, ?, ?], welche die Verwendung von APIs im Allgemeinen betrachten. Dabei werden existierende Verwendungsszenarien analysiert, um Muster im Umgang mit der Programmierschnittstelle wie beispielsweise Abhängigkeiten zu anderen Elementen zu eruieren(vgl. [?]), häufig bzw. wenig genutzte Features der API herauszustellen (vgl. [?]) oder aus diesen Spezifikation abzuleiten (vgl. [?]). Es sind Kongruenzen zwischen einzelnen Bestandteilen dieser Analysen und Untersuchungen dieser Arbeit zu beobachten, jedoch existiert ein grundsätzlicher Unterschied hinsichtlich der Definition, was die Verwendung einer API umfasst. Während in den angeführten Untersuchungen allgemeine Betrachtungen angestellt werden, steht in dieser Thesis der Gebrauch von Frameworks in Abgrenzung zur Verwendung als Library im Vordergrund. Dementsprechend beinhaltet diese Analyse Aspekte, die über die reine *usage analysis* hinausgehen und sich dem Aufbau von Programmierschnittstellen widmen. Es findet demnach eine Spezialisierung statt, um gezielter spezifische Muster zu erkennen.

Objekt–orientierte Frameworks Gerade diese Eigenschaften von Frameworks sind Bestandteil diverse Veröffentlichungen. Grundlegende Untersuchung wie [?] und [?] gehen auf die konkrete Anwendung von Frameworks zur Wiederverwendung von

Klassen und Anwendungssystemen ein und beschreiben erste Eigenschaften von solchen Programmierschnittstellen, wie die Bereitstellung abstrakter Features und die Notwendigkeit einer Instanziierung durch Anwendungscode. Darauf aufbauend wird in [**?**] an Hand von Refactoring–Maßnahmen für ausgewählte Beispiele mögliche Verbesserungen am Design aufgezeigt, um durch Generalisierungen ein höheres Level an Abstraktheit, sowie durch Spezialisierungen ein breitgefächertes Objektmodell zu erzielen und somit das Verständnis des Frameworks zu verbessern. In [**?**] werden Probleme und Erfahrungen im Umgang mit derartigen APIs beschrieben und analysiert. Die Schlussfolgerungen dieser Untersuchung heben wiederum das Verständnis von Frameworks als wichtigen Aspekt zur unproblematischen Verwendung hervor. Die in dieser Arbeit beschrieben Analyse setzt an dieser Problematik an und eröffnet neue Ansätze zur besseren Kenntnis von Frameworks.

Kapitel 7

Ausblick

Resümierend betrachtet sind einige Punkte und Aspekte der vorgestellten Untersuchung als problematisch anzusehen.

Projektauswahl So ist zum einen die Projektauswahl zu diskutieren, weniger in qualitativer als vielmehr in quantitativer Hinsicht. Einerseits liefert der kleine Corpus an ausgewählten Projekten im Vergleich zum groß angelegten Ansatz in [?] teilweise bessere Ergebnisse, jedoch lassen sich viele APIs nicht bewertet. Dieser Umstand gründet vor allem in der Tatsache, dass für viele Programmierschnittstellen die Anzahl an Projekten, die diese im Sinne eines Frameworks nutzen, nicht repräsentativ ist. Die Aussagen, die getätigt werden können, optimieren vorangegangene Analysen, jedoch ist der Verlust anderer interessanter Aspekte bei dieser Vorgehensweise denkbar. Ein wichtiger Ansatz zur Verbesserung dieser Ausarbeitung liegt also in einem größeren, jedoch ähnlich strukturierten Corpus. Dieser Punkt ist vom Software Languages Team bereits initialisiert, indem auf bestehende Corpus–Projekte wie beispielsweise dem „Qualitas"–Corpus der Universität Auckland[1] eingegangen wird.

API–Auswahl Ebenso ist die API–Auswahl einer Verbesserung zu unterziehen, da die genutzte Umsetzung davon ausgeht, dass jedes JAR–Archiv, welches im Projekt mitgeliefert oder während des Build–Vorgangs heruntergeladen wird, als Library Verwendung findet. Der API–Pool ist demnach zu überarbeiten, indem entweder der Kompilierungsprozess der Projekte gezielter analysiert wird oder weitere Corpora als Vergleichsobjekte untersucht werden, um die bisherige Auswahl mit den gewonnen Erkenntnissen abzugleichen.

[1] vgl. http://qualitascorpus.com/

Korrektheit der Tools Des Weiteren stützt sich diese Analyse darauf, dass die Umgebung einwandfrei und korrekt funktioniert. Da dies noch nicht verifiziert wurde, besteht eine wichtige Aufgabe darin *senity checks* durchzuführen, um einerseits zu gewährleisten, dass die beiden Tools (vgl. **??**) ihre Aufgaben korrekt erfüllen, andererseits jedoch auch kompatibel zu einander sind und in dem Sinne gleich „mächtig“ sind, dass sie für äquivalente Eingaben auch ähnliche Ausgaben erzeugen. Hierbei sind gezwungenermaßen Einschränkungen hinzunehmen, da sich Sourcecode und Bytecode beispielsweise auf Grund von Optimierungen durch den Compiler nicht äquivalent entsprechen. Diese Thematik ist derzeit bereits ebenfalls Bestandteil der aktuellen Forschungsarbeit des Software Languages Team.

Abweichende Framework–Ansätze Letztendlich bietet auch die vorgestellte und umgesetzte Analyse noch Möglichkeiten zur Verbesserung und Ergänzung. Im Vordergrund steht hier die Einbindung anderer Ansätze für Frameworks, wie es in verschiedenen Abschnitten bereits diskutiert wurde (vgl. **??**). Dazu muss eine Möglichkeit zur Verarbeitung von Annotations und XML–Konfigurationen geschaffen werden, aber auch spezifiziert werden, wie Framework–Verhalten in diesen Fällen definiert ist.

Weitere Framework–Aspekte Darüber hinaus sind Methodenerweiterungen als wichtiger Aspekt von Frameworks in dieser Untersuchung zwar enthalten, haben jedoch keine Relevanz hinsichtlich der Bewertung von Framework–Eigenschaften. Dazu müsste die Analyse erweitert werden, um beispielsweise ebenfalls „framework–hafte“ Erweiterungen im Bereich der Methoden festzustellen, aber auch abgeleitete Methoden erkennen, deren einzige Funktionalität dem Aufruf der Supermethode entspricht.

Kapitel 8

Fazit

Zusammenfassend ist festzuhalten, dass bestimmt werden kann, ob eine API als Framework Verwendung findet. Dies umfasst dabei diverse Domänen und verschiedene Arten von APIs, wobei die dargestellten Merkmale augenscheinlich für Anwendungen aus den Bereichen XML und GUI signifikant sind. Wie ausgeführt impliziert das Auftreten von Erweiterungen und Implementierungen keineswegs, dass eine Programmierschnittstelle viele Typen bereitstellt, die einen derartigen Gebrauch ermöglichen, da sich die *Frameworkiness* einer API auf einzelne Bereiche beschränken kann wie beispielsweise `AWT` mit den Features für die Ereignisbehandlung. Insbesondere verhält es sich sogar so, dass die Verwendung in Projekten stark einzelne Komponenten fokussiert, d. h. also Typen in einem einzelnen Projekt mehrfach aus bestimmten Elementen abgeleitet werden. Somit ist das Vorkommen von API–Typen ohne konkreten Subtyp innerhalb der Programmierschnittstelle selbst kein Indiz für eine vermehrte Verwendung als Framework, da die *subtyping requiredness* im Anwendungscode keine Erweiterungen oder Implementierungen erzwingt. Sie erfordert aber, dass eine Implementation vorhanden sein muss, falls die entsprechen Programmierschnittstelle genutzt werden soll, was jedoch gleichfalls durch Provider gewährleistet sein kann. Dies geschieht im Beispiel von `DOM` und `StAX` versteckt, da in diesem Fall diese Provider–Implementationen von der `JAXP` für den Anwender nicht sichtbar ausgewählt werden. Dementsprechend ist die Verwendung eines Frameworks keineswegs evident, kann aber durch gezielte Analysen aufgedeckt werden.

Kapitel 9

Danksagungen

Abschließend möchte ich dem gesamten „APIAnalysis2.0“–Team für die Unterstützung danken, insbesondere Prof. Dr. Ralf Lämmel, Ekatarina Pek, Sebastian Jackel, David Klauer, Malte Knauf und Joachim Pehl. Viele interessante Ideen und hilfreiche Kritikpunkte, die in diese Arbeit mit eingeflossen sind, entsprangen den regelmäßigen Meetings. Hinzu kommen die Tools der Analyseumgebung, die vom Team bereitgestellt oder mit dessen Hilfe entwickelt wurden.

Anhang A

Provider–Analyse

A.1 Konzeptioneller Weg mit Provider–Analyse

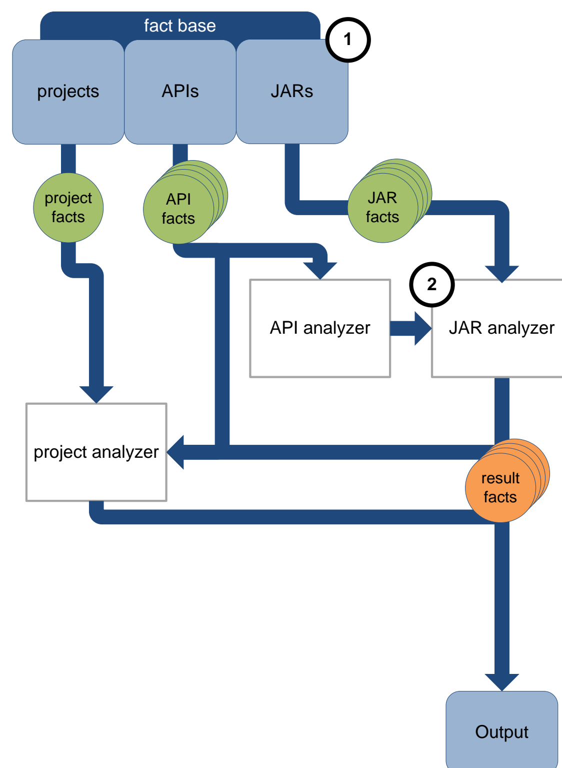

Abbildung A.1: Konzeptioneller Weg mit Provider–Analyse

A.2 Messgrößen der Provider–Analyse

```
% by_jar_ext_abstr_class(Jid,Aid,Package,Name,Supertypes,Modifiers)
% jar extends non-subtyped class
by_jar_ext_abstr_class(Jid,Aid,P,N,Ss,Ms) :-
    by_all_unext_abstr_class(Aid,P,N,Ss,Ms),
    full_name(P, N, Full),
    % concrete jar class extends non-subtyped
    class(Jid,_,_,Full,_,Ms_j),
    \+ member('abstract', Ms_j).

% by_jar_impl_interface(Jid,Aid,Package,Name,Supertypes,Modifiers)
% jar implements non-subtyped interface
by_jar_impl_interface(Jid,Aid,P,N,Ss,Ms) :-
    by_all_unimpl_interface(Aid,P,N,Ss,Ms),
    full_name(P, N, Full),
    % concrete jar interface implements non-subtyped
    class(Jid,_,_,_,Is_j,Ms_j),
    member(Full, Is_j),
    \+ member('abstract', Ms_j).
```

Anhang B

Analyse–Umgebung

B.1 Corpus „JavaShape“

Tabelle B.1: Liste aller Projekte

B.2 API–Pool

Tabelle B.2: Liste aller APIs